Custom EBook 2

Written by Jainish Parmar

Table of Contents

5.6 Further Reading

As its name suggests, YACC was not the first compiler construction tool, but it remains widely used today and has led to a proliferation of similar tools written in various languages and addressing different classes of grammars. Here is just a small selection:

1. S. C. Johnson, "YACC: Yet Another Compiler-Compiler", Bell Laboratories Technical Journal, 1975.

2. D. Grune and C.J.H Jacobs, "A programmer-friendly LL(1) parser generator", Software: Practice and Experience, volume 18, number 1.

3. T.J. Parr and R.W. Quong, "ANTLR: A predicated LL(k) Parser Generator", Software: Practice and Experience, 1995.

4. S. McPeak, G.C. Necula, "Elkhound: A Fast, Practical GLR Parser Generator", International Conference on Compiler Construction, 2004.

Chapter 6 – The Abstract Syntax Tree

6.1 Overview

The **abstract syntax tree (AST)** is an important internal data structure that represents the primary structure of a program. The AST is the starting point for semantic analysis of a program. It is "abstract" in the sense that the structure leaves out the particular details of parsing: the AST does not care whether a language has prefix, postfix, or infix expressions. (In fact, the AST we describe here can be used to represent most procedural languages.)

For our project compiler, we will define an AST in terms of five C structures representing declarations, statements, expressions, types, and parameters. While you have certainly encountered each of these terms while learning programming, they are not always used precisely in practice. This chapter will help you to sort those items out very clearly:

- A **declaration** states the name, type, and value of a symbol so that it can be used in the program. Symbols include items such as constants, variables, and functions.
- A **statement** indicates an action to be carried out that changes the state of the program. Examples include loops, conditionals, and function returns.
- An **expression** is a combination of values and operations that is **evaluated** according to specific rules and yields a **value** such as an integer, floating point, or string. In some programming languages, an expression may also have a **side effect** that changes the state of the program.

For each kind of element in the AST, we will give an example of the code and how it is constructed. Because each of these structures potentially has pointers to each of the other types, it is necessary to preview all of them before seeing how they work together.

Once you understand all of the elements of the AST, we finish the chapter by demonstrating how the entire structure can be created automatically through the use of the Bison parser generator.

6.2 Declarations

A complete B-Minor program is a sequence of declarations. Each declaration states the existence of a variable or a function. A variable declaration may optionally give an initializing value. If none is given, it is given a default value of zero. A function declaration may optionally give the body of the function in code; if no body is given, then the declaration serves as a prototype for a function declared elsewhere.

For example, the following are all valid declarations:

```
b: boolean;
s: string = "hello";
f: function integer ( x: integer ) = { return x*x; }
```

A declaration is represented by a `decl` structure that gives the name, type, value (if an expression), code (if a function), and a pointer to the next declaration in the program:

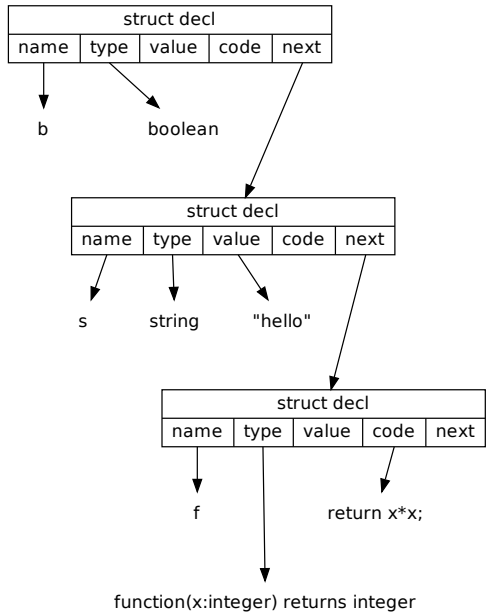
```
struct decl {
    char *name;
    struct type *type;
    struct expr *value;
    struct stmt *code;
    struct decl *next;
};
```

Because we will be creating a lot of these structures, you will need a factory function that allocates a structure and initializes its fields, like this:

```
struct decl * decl_create( char *name,
                           struct type *type,
                           struct expr *value,
                           struct stmt *code,
                           struct decl *next )
{
    struct decl *d = malloc(sizeof(*d));
    d->name = name;
    d->type = type;
    d->value = value;
    d->code = code;
    d->next = next;
    return d;
}
```

(You will need to write similar code for statements, expressions, etc, but we won't keep repeating it here.)

The three declarations on the preceding page can be represented graphically as a linked list, like this:



Note that some of the fields point to nothing: these would be represented by a null pointer, which we omit for clarity. Also, our picture is incomplete and must be expanded: the items representing types, expressions, and statements are all complex structures themselves that we must describe.

6.3 Statements

The body of a function consists of a sequence of statements. A statement indicates that the program is to take a particular action in the order specified, such as computing a value, performing a loop, or choosing between branches of an alternative. A statement can also be a declaration of a local variable. Here is the `stmt` structure:

```
struct stmt {
    stmt_t kind;
    struct decl *decl;
    struct expr *init_expr;
    struct expr *expr;
    struct expr *next_expr;
    struct stmt *body;
    struct stmt *else_body;
    struct stmt *next;
};
```

```
typedef enum {
    STMT_DECL,
    STMT_EXPR,
    STMT_IF_ELSE,
    STMT_FOR,
    STMT_PRINT,
    STMT_RETURN,
    STMT_BLOCK
} stmt_t;
```

The `kind` field indicates what kind of statement it is:

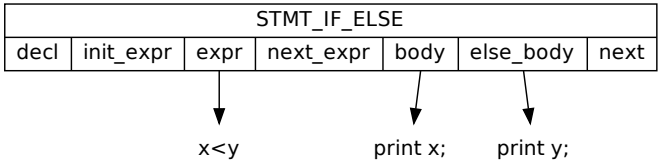
- `STMT_DECL` indicates a (local) declaration, and the `decl` field will point to it.
- `STMT_EXPR` indicates an expression statement and the `expr` field will point to it.
- `STMT_IF_ELSE` indicates an if-else expression such that the `expr` field will point to the control expression, the `body` field to the statements executed if it is true, and the `else_body` field to the statements executed if it is false.
- `STMT_FOR` indicates a for-loop, such that `init_expr`, `expr`, and `next_expr` are the three expressions in the loop header, and `body` points to the statements in the loop.
- `STMT_PRINT` indicates a `print` statement, and `expr` points to the expressions to print.
- `STMT_RETURN` indicates a `return` statement, and `expr` points to the expression to return.
- `STMT_BLOCK` indicates a block of statements inside curly braces, and `body` points to the contained statements.

And, as we did with declarations, we require a function `stmt_create` to create and return a statement structure:

```
struct stmt * stmt_create( stmt_t kind,
    struct decl *decl, struct expr *init_expr,
    struct expr *expr, struct expr *next_expr,
    struct stmt *body, struct stmt *else_body,
    struct stmt *next );
```

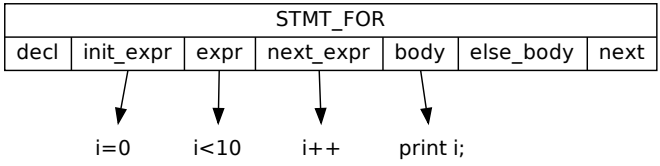
This structure has a lot of fields, but each one serves a purpose and is used when necessary for a particular kind of statement. For example, an if-else statement only uses the `expr`, `body`, and `else_body` fields, leaving the rest null:

```
if( x<y ) print x; else print y;
```



A for-loop uses the three `expr` fields to represent the three parts of the loop control, and the `body` field to represent the code being executed:

```
for(i=0;i<10;i++) print i;
```



6.4 Expressions

Expressions are implemented much like the simple expression AST shown in Chapter 5. The difference is that we need many more binary types: one for every operator in the language, including arithmetic, logical, comparison, assignment, and so forth. We also need one for every type of leaf value, including variable names, constant values, and so forth. The `name` field will be set for `EXPR_NAME`, the `integer_value` field for `EXPR_INTEGER_LITERAL`, and so on. You may need to add values and types to this structure as you expand your compiler.

```
struct expr {
    expr_t kind;
    struct expr *left;
    struct expr *right;

    const char *name;
    int integer_value;
    const char *
       string_literal;
};
```

```
typedef enum {
    EXPR_ADD,
    EXPR_SUB,
    EXPR_MUL,
    EXPR_DIV,
    ...
    EXPR_NAME,
    EXPR_INTEGER_LITERAL,
    EXPR_STRING_LITERAL
} expr_t;
```

As before, you should create a factory for a binary operator:

```
struct expr * expr_create( expr_t kind,
                           struct expr *L,
                           struct expr *R );
```

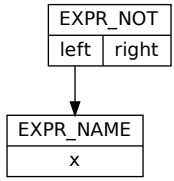
And then a factory for each of the leaf types:

```
struct expr * expr_create_name( const char *name );
struct expr * expr_create_integer_literal( int i );
struct expr * expr_create_boolean_literal( int b );
struct expr * expr_create_char_literal( char c );
struct expr * expr_create_string_literal
                               ( const char *str );
```

Note that you can store the integer, boolean, and character literal values all in the `integer_value` field.

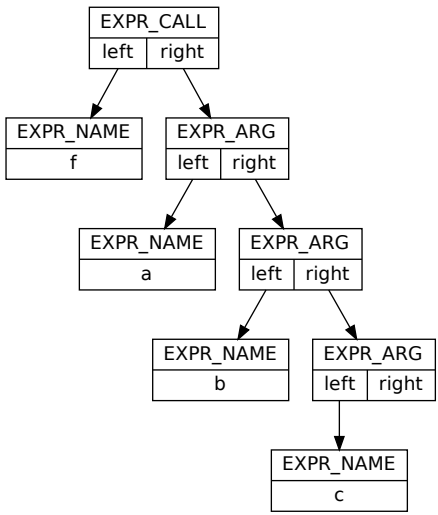
A few cases deserve special mention. Unary operators like logical-not typically have their sole argument in the `left` pointer:

```
!x
```



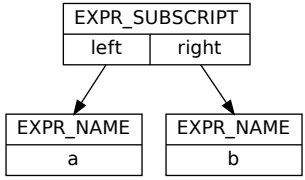
A function call is constructed by creating an `EXPR_CALL` node, such that the left-hand side is the function name, and the right hand side is an unbalanced tree of `EXPR_ARG` nodes. While this looks a bit awkward, it allows us to express a linked list using a tree, and will simplify the handling of function call arguments on the stack during code generation.

```
f(a,b,c)
```



Array subscripting is treated like a binary operator, such that the name of the array is on the left side of the `EXPR_SUBSCRIPT` operator, and an integer expression on the right:

```
a[b]
```



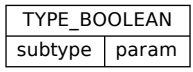
6.5 Types

A `type` structure encodes the type of every variable and function mentioned in a declaration. Primitive types like `integer` and `boolean` are expressed by simply setting the `kind` field appropriately, and leaving the other fields null. Compound types like `array` and `function` are built by connecting multiple `type` structures together.

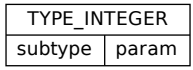
```
typedef enum {
    TYPE_VOID,
    TYPE_BOOLEAN,
    TYPE_CHARACTER,
    TYPE_INTEGER,
    TYPE_STRING,
    TYPE_ARRAY,
    TYPE_FUNCTION
} type_t;

struct type {
    type_t kind;
    struct type *subtype;
    struct param_list *params;
};

struct param_list {
    char *name;
    struct type *type;
    struct param_list *next;
};
```

For example, to express a basic type like a boolean or an integer, we simply create a standalone `type` structure, with `kind` set appropriately, and the other fields null:

`boolean`

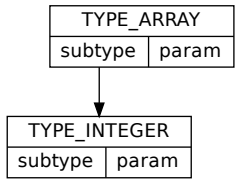


`integer`



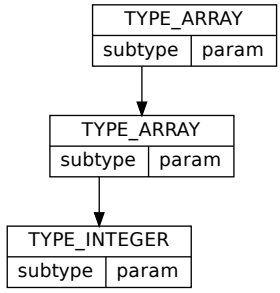
To express a compound type like an array of integers, we set `kind` to `TYPE_ARRAY` and set `subtype` to point to a `TYPE_INTEGER`:

`array [] integer`



These can be linked to arbitrary depth, so to express an array of array of integers:

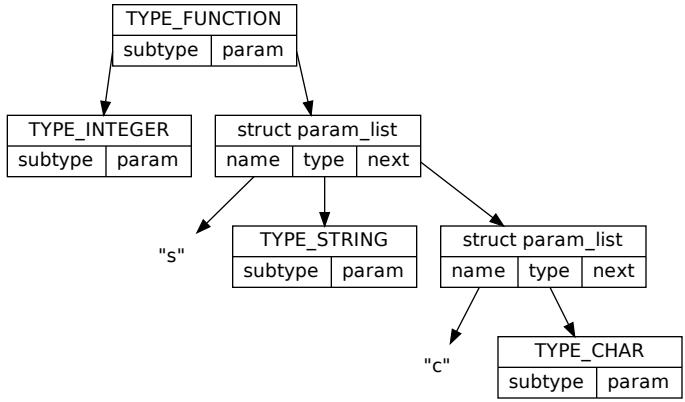
`array [] array [] integer`



To express the type of a function, we use `subtype` to express the return type of the function, and then connect a linked list of `param_list` nodes to describe the name and type of each parameter to the function.

For example, here is the type of a function which takes two arguments and returns an integer:

```
function integer (s:string, c:char)
```



Note that the type structures here let us express some complicated and powerful higher order concepts of programming. By simply swapping in complex types, you can describe an array of ten functions, each returning an integer:

```
a: array [10] function integer ( x: integer );
```

Or how about a function that returns a function?

```
f: function function integer (x:integer) (y:integer);
```

Or even a function that returns an array of functions!

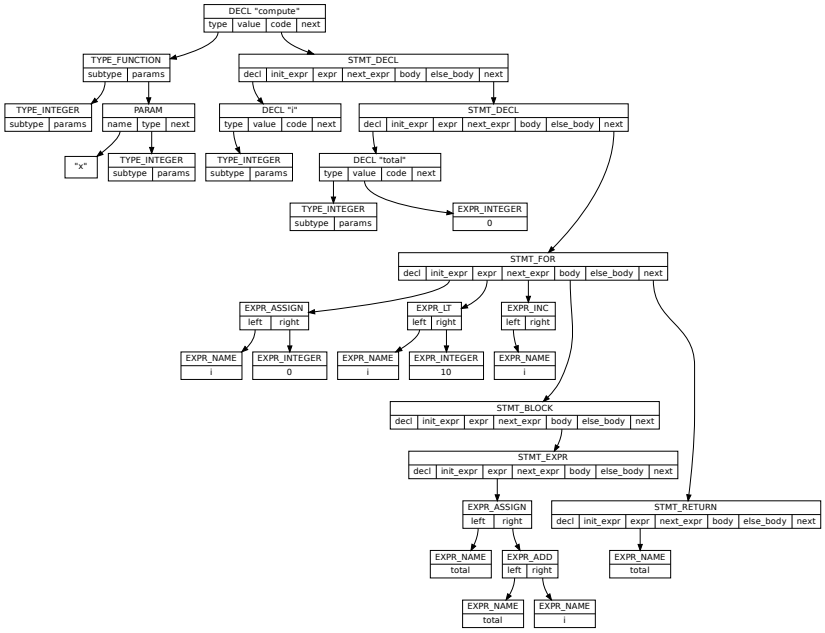
```
g: function array [10]
       function integer (x:integer) (y:integer);
```

While the B-Minor type system is capable of *expressing* these ideas, these combinations will be rejected later in typechecking, because they require a more dynamic implementation than we are prepared to create. If you find these ideas interesting, then you should read up on functional languages such as Scheme and Haskell.

6.6 Putting it All Together

Now that you have seen each individual component, let's see how a complete B-Minor function would be expressed as an AST:

```
compute: function integer ( x:integer ) = {
        i: integer;
        total: integer = 0;
        for(i=0;i<10;i++) {
                total = total + i;
        }
        return total;
}
```



6.7 Building the AST

With the functions created so far in this chapter, we could, in principle, construct the AST manually in a sort of nested style. For example, the following code represents a function called `square` which accepts an integer `x` as a parameter, and returns the value `x*x`:

```
d = decl_create(
    "square",
    type_create(TYPE_FUNCTION,
        type_create(TYPE_INTEGER,0,0),
        param_list_create(
            "x",
            type_create(TYPE_INTEGER,0,0),
            0)),
    0,
    stmt_create(STMT_RETURN,0,0,
        expr_create(EXPR_MUL,
            expr_create_name("x"),
            expr_create_name("x")),
        0,0,0,0),
    0);
```

Obviously, this is no way to write code! Instead, we want our parser to invoke the various creation functions whenever they are reduced, and then hook them up into a complete tree. Using an LR parser generator like Bison, this process is straightforward. (Here I will give you the idea of how to proceed, but you will need to figure out many of the details in order to complete the parser.)

At the top level, a B-Minor program is a sequence of declarations:

```
program : decl_list
          { parser_result = $1; }
        ;
```

Then, we write rules for each of the various kinds of declarations in a B-Minor program:

```
decl : name TOKEN_COLON type TOKEN_SEMI
     { $$ = decl_create($1,$3,0,0,0); }
   | name TOKEN_COLON type TOKEN_ASSIGN expr TOKEN_SEMI
     { $$ = decl_create($1,$3,$5,0,0); }
   | /* and more cases here */
       . . .
   ;
```

Since each `decl` structure is created separately, we must connect them together in a linked list formed by a `decl_list`. This is most easily done by making the rule right-recursive, so that `decl` on the left represents one declaration, and `decl_list` on the right represents the remainder of the linked list. The end of the list is a null value when `decl_list` produces ϵ.

```
decl_list : decl decl_list
            { $$ = $1; $1->next = $2; }
          | /* epsilon */
            { $$ = 0; }
          ;
```

For each kind of statement, we create a `stmt` structure that pulls out the necessary elements from the grammar.

```
stmt : TOKEN_IF TOKEN_LPAREN expr TOKEN_RPAREN stmt
       { $$ = stmt_create(STMT_IF_ELSE,0,0,$3,0,$5,0,0); }
     | TOKEN_LBRACE stmt_list TOKEN_RBRACE
       { $$ = stmt_create(STMT_BLOCK,0,0,0,0,$2,0,0); }
     | /* and more cases here */
       . . .
     ;
```

Proceed in this way down through each of the grammar elements of a B-Minor program: declarations, statements, expressions, types, parameters, until you reach the leaf elements of literal values and symbols, which are handled in the same way as in Chapter 5.

There is one last complication: What, exactly is the semantic type of the values returned as each rule is reduced? It isn't a single type, because each kind of rule returns a different data structure: a declaration rule returns a `struct decl *`, while an identifier rule returns a `char *`. To make this work, we inform Bison that the semantic value is the union of all of the types in the AST:

```
%union {
        struct decl *decl;
        struct stmt *stmt;
        . . .
        char *name;
};
```

And then indicate the specific subfield of the union used by each rule:

```
%type <decl> program decl_list decl . . .
%type <stmt> stmt_list stmt . . .
. . .
%type <name> name
```

6.8 Exercises

1. Write a complete LR grammar for B-Minor and test it using Bison. Your first attempt will certainly have many shift-reduce and reduce-reduce conflicts, so use your knowledge of grammars from Chapter 4 to rewrite the grammar and eliminate the conflicts.

2. Write the AST structures and generating functions as outlined in this chapter, and manually construct some simple ASTs using nested function calls as shown above.

3. Add new functions `decl_print()`, `stmt_print()`, etc. that print the AST back out so you can verify that the program was generated correctly. Make your output nicely formatted using indentation and consistent spacing, so that the code is easily readable.

4. Add the AST generator functions as action rules to your Bison grammar, so that you can parse complete programs, and print them back out again.

5. Add new functions `decl_translate()`, `stmt_translate()`, etc. that output the B-Minor AST in a different language of your own choosing, such as Python or Java or Rust.

6. Add new functions that emit the AST in a graphical form so you can "see" the structure of a program. One approach would be to use the Graphviz DOT format: let each declaration, statement, etc be a node in a graph, and then let each pointer between structures be an edge in the graph.

Chapter 7 – Semantic Analysis

Now that we have completed construction of the AST, we are ready to begin analyzing the **semantics**, or the actual meaning of a program, and not simply its structure.

Type checking is a major component of semantic analysis. Broadly speaking, the type system of a programming language gives the programmer a way to make verifiable assertions that the compiler can check automatically. This allows for the detection of errors at compile-time, instead of at runtime.

Different programming languages have different approaches to type checking. Some languages (like C) have a rather weak type system, so it is possible to make serious errors if you are not careful. Other languages (like Ada) have very strong type systems, but this makes it more difficult to write a program that will compile at all!

Before we can perform type checking, we must determine the type of each identifier used in an expression. However, the mapping between variable names and their actual storage locations is not immediately obvious. A variable `x` in an expression could refer to a local variable, a function parameter, a global variable, or something else entirely. We solve this problem by performing **name resolution**, in which each definition of a variable is entered into a **symbol table**. This table is referenced throughout the semantic analysis stage whenever we need to evaluate the correctness of some code.

Once name resolution is completed, we have all the information necessary to check types. In this stage, we compute the type of complex expressions by combining the basic types of each value according to standard conversion rules. If a type is used in a way that is not permitted, the compiler will output an (ideally helpful) error message that will assist the programmer in resolving the problem.

Semantic analysis also includes other forms of checking the correctness of a program, such as examining the limits of arrays, avoiding bad pointer traversals, and examining control flow. Depending on the design of the language, some of these problems can be detected at compile time, while others may need to wait until runtime.

7.1 Overview of Type Systems

Most programming languages assign to every value (whether a literal, constant, or variable) a **type**, which describes the interpretation of the data in that variable. The type indicates whether the value is an integer, a floating point number, a boolean, a string, a pointer, or something else. In most languages, these atomic types can be combined into higher-order types such as enumerations, structures, and variant types to express complex constraints.

The type system of a language serves several purposes:

- **Correctness.** A compiler uses type information provided by the programmer to raise warnings or errors if a program attempts to do something improper. For example, it is almost certainly an error to assign an integer value to a pointer variable, even though both might be implemented as a single word in memory. A good type system can help to eliminate runtime errors by flagging them at compile time instead.

- **Performance.** A compiler can use type information to find the most efficient implementation of a piece of code. For example, if the programmer tells the compiler that a given variable is a constant, then the same value can be loaded into a register and used many times, rather than constantly loading it from memory.

- **Expressiveness.** A program can be made more compact and expressive if the language allows the programmer to leave out facts that can be inferred from the type system. For example, in B-Minor the `print` statement does not need to be told whether it is printing an integer, a string, or a boolean: the type is inferred from the expression and the value is automatically displayed in the proper way.

A programming language (and its type system) are commonly classified on the following axes:

- safe or unsafe
- static or dynamic
- explicit or implicit

In an **unsafe programming language**, it is possible to write valid programs that have wildly undefined behavior that violates the basic structure of the program. For example, in the C programming language, a program can construct an arbitrary pointer to modify any word in memory, and thereby change the data and code of the compiled program. Such power is probably necessary to implement low-level code like an operating system or a driver, but is problematic in general application code.

For example, the following code in C is syntactically legal and will compile, but is unsafe because it writes data outside the bounds of the array `a[]`. As a result, the program could have almost any outcome, including incorrect output, silent data corruption, or an infinite loop.

```
/* This is C code */
int i;
int a[10];
for(i=0;i<100;i++) a[i] = i;
```

In a **safe programming language**, it is not possible to write a program that violates the basic structures of the language. That is, no matter what input is given to a program written in a safe language, it will always execute in a well-defined way that preserves the abstractions of the language. A safe programming language enforces the boundaries of arrays, the use of pointers, and the assignment of types to prevent undefined behavior. Most interpreted languages, like Perl, Python, and Java, are safe languages.

For example, in C#, the boundaries of arrays are checked at runtime, so that running off the end of an array has the predictable effect of throwing an `IndexOutOfRangeException`:

```
/* This is C-sharp code */
a = new int[10];
for(int i=0;i<100;i++) a[i] = i;
```

In a **statically typed language**, all typechecking is performed at compile-time, long before the program runs. This means that the program can be translated into basic machine code without retaining any of the type information, because all operations have been checked and determined to be safe. This yields the most high performance code, but it does eliminate some kinds of convenient programming idioms.

Static typing is often used to distinguish between integer and floating point operations. While operations like addition and multiplication are usually represented by the same symbols in the source language, they are implemented with fundamentally different machine code. For example, in the C language on X86 machines, `(a+b)` would be translated to an `ADDL` instruction for integers, but an `FADD` instruction for floating point values. To know which instruction to apply, we must first determine the type of `a` and `b` and deduce the intended meaning of `+`.

In a **dynamically typed language**, type information is available at runtime and stored in memory alongside the data that it describes. As the program executes, the safety of each operation is checked by comparing the types of each operand. If types are observed to be incompatible, then the program must halt with a runtime type error. This also allows for

correct, Trudy now knows when she got it right and when she got it wrong. In Fig. 8-5, she got it right for bits 0, 1, 2, 3, 4, 6, 8, 12, and 13. But she knows from Alice's reply in Fig. 8-5(d) that only bits 1, 3, 7, 8, 10, 11, 12, and 14 are part of the one-time pad. For four of these bits (1, 3, 8, and 12), she guessed right and captured the correct bit. For the other four (7, 10, 11, and 14), she guessed wrong and does not know the bit transmitted. Thus, Bob knows the one-time pad starts with 01011001, from Fig. 8-5(e) but all Trudy has is 01?1??0?, from Fig. 8-5(g).

Of course, Alice and Bob are aware that Trudy may have captured part of their one-time pad, so they would like to reduce the information Trudy has. They can do this by performing a transformation on it. For example, they could divide the one-time pad into blocks of 1024 bits, square each one to form a 2048-bit number, and use the concatenation of these 2048-bit numbers as the one-time pad. With her partial knowledge of the bit string transmitted, Trudy has no way to generate its square and so has nothing. The transformation from the original one-time pad to a different one that reduces Trudy's knowledge is called **privacy amplification**. In practice, complex transformations in which every output bit depends on every input bit are used instead of squaring.

Poor Trudy. Not only does she have no idea what the one-time pad is, but her presence is not a secret either. After all, she must relay each received bit to Bob to trick him into thinking he is talking to Alice. The trouble is, the best she can do is transmit the qubit she received, using the polarization she used to receive it, and about half the time she will be wrong, causing many errors in Bob's one-time pad.

When Alice finally starts sending data, she encodes it using a heavy forward-error-correcting code. From Bob's point of view, a 1-bit error in the one-time pad is the same as a 1-bit transmission error. Either way, he gets the wrong bit. If there is enough forward error correction, he can recover the original message despite all the errors, but he can easily count how many errors were corrected. If this number is far more than the expected error rate of the equipment, he knows that Trudy has tapped the line and can act accordingly (e.g., tell Alice to switch to a radio channel, call the police, etc.). If Trudy had a way to clone a photon so she had one photon to inspect and an identical photon to send to Bob, she could avoid detection, but at present no way to clone a photon perfectly is known. And even if Trudy could clone photons, the value of quantum cryptography to establish one-time pads would not be reduced.

Although quantum cryptography has been shown to operate over distances of 60 km of fiber, the equipment is complex and expensive. Still, the idea has promise. For more information about quantum cryptography, see Mullins (2002).

8.1.5 Two Fundamental Cryptographic Principles

Although we will study many different cryptographic systems in the pages ahead, two principles underlying all of them are important to understand. Pay attention. You violate them at your peril.

Redundancy

The first principle is that all encrypted messages must contain some redundancy, that is, information not needed to understand the message. An example may make it clear why this is needed. Consider a mail-order company, The Couch Potato (TCP), with 60,000 products. Thinking they are being very efficient, TCP's programmers decide that ordering messages should consist of a 16-byte customer name followed by a 3-byte data field (1 byte for the quantity and 2 bytes for the product number). The last 3 bytes are to be encrypted using a very long key known only by the customer and TCP.

At first, this might seem secure, and in a sense it is because passive intruders cannot decrypt the messages. Unfortunately, it also has a fatal flaw that renders it useless. Suppose that a recently fired employee wants to punish TCP for firing her. Just before leaving, she takes the customer list with her. She works through the night writing a program to generate fictitious orders using real customer names. Since she does not have the list of keys, she just puts random numbers in the last 3 bytes, and sends hundreds of orders off to TCP.

When these messages arrive, TCP's computer uses the customers' name to locate the key and decrypt the message. Unfortunately for TCP, almost every 3-byte message is valid, so the computer begins printing out shipping instructions. While it might seem odd for a customer to order 837 sets of children's swings or 540 sandboxes, for all the computer knows, the customer might be planning to open a chain of franchised playgrounds. In this way, an active intruder (the ex-employee) can cause a massive amount of trouble, even though she cannot understand the messages her computer is generating.

This problem can be solved by the addition of redundancy to all messages. For example, if order messages are extended to 12 bytes, the first 9 of which must be zeros, this attack no longer works because the ex-employee can no longer generate a large stream of valid messages. The moral of the story is that all messages must contain considerable redundancy so that active intruders cannot send random junk and have it be interpreted as a valid message.

However, adding redundancy makes it easier for cryptanalysts to break messages. Suppose that the mail-order business is highly competitive, and The Couch Potato's main competitor, The Sofa Tuber, would dearly love to know how many sandboxes TCP is selling so it taps TCP's phone line. In the original scheme with 3-byte messages, cryptanalysis was nearly impossible because after guessing a key, the cryptanalyst had no way of telling whether it was right because almost every message was technically legal. With the new 12-byte scheme, it is easy for the cryptanalyst to tell a valid message from an invalid one. Thus, we have

Cryptographic principle 1: Messages must contain some redundancy

In other words, upon decrypting a message, the recipient must be able to tell whether it is valid by simply inspecting the message and perhaps performing a

simple computation. This redundancy is needed to prevent active intruders from sending garbage and tricking the receiver into decrypting the garbage and acting on the "plaintext." However, this same redundancy makes it much easier for passive intruders to break the system, so there is some tension here. Furthermore, the redundancy should never be in the form of *n* 0s at the start or end of a message, since running such messages through some cryptographic algorithms gives more predictable results, making the cryptanalysts' job easier. A CRC polynomial is much better than a run of 0s since the receiver can easily verify it, but it generates more work for the cryptanalyst. Even better is to use a cryptographic hash, a concept we will explore later. For the moment, think of it as a better CRC.

Getting back to quantum cryptography for a moment, we can also see how redundancy plays a role there. Due to Trudy's interception of the photons, some bits in Bob's one-time pad will be wrong. Bob needs some redundancy in the incoming messages to determine that errors are present. One very crude form of redundancy is repeating the message two times. If the two copies are not identical, Bob knows that either the fiber is very noisy or someone is tampering with the transmission. Of course, sending everything twice is overkill; a Hamming or Reed-Solomon code is a more efficient way to do error detection and correction. But it should be clear that some redundancy is needed to distinguish a valid message from an invalid message, especially in the face of an active intruder.

Freshness

The second cryptographic principle is that measures must be taken to ensure that each message received can be verified as being fresh, that is, sent very recently. This measure is needed to prevent active intruders from playing back old messages. If no such measures were taken, our ex-employee could tap TCP's phone line and just keep repeating previously sent valid messages. Thus,

Cryptographic principle 2: Some method is needed to foil replay attacks

One such measure is including in every message a timestamp valid only for, say, 10 seconds. The receiver can then just keep messages around for 10 seconds and compare newly arrived messages to previous ones to filter out duplicates. Messages older than 10 seconds can be thrown out, since any replays sent more than 10 seconds later will be rejected as too old. Measures other than timestamps will be discussed later.

8.2 SYMMETRIC-KEY ALGORITHMS

Modern cryptography uses the same basic ideas as traditional cryptography (transposition and substitution), but its emphasis is different. Traditionally, cryptographers have used simple algorithms. Nowadays, the reverse is true: the object

is to make the encryption algorithm so complex and involuted that even if the cryptanalyst acquires vast mounds of enciphered text of his own choosing, he will not be able to make any sense of it at all without the key.

The first class of encryption algorithms we will study in this chapter are called **symmetric-key algorithms** because they use the same key for encryption and decryption. Fig. 8-2 illustrates the use of a symmetric-key algorithm. In particular, we will focus on **block ciphers**, which take an n-bit block of plaintext as input and transform it using the key into an n-bit block of ciphertext.

Cryptographic algorithms can be implemented in either hardware (for speed) or software (for flexibility). Although most of our treatment concerns the algorithms and protocols, which are independent of the actual implementation, a few words about building cryptographic hardware may be of interest. Transpositions and substitutions can be implemented with simple electrical circuits. Figure 8-6(a) shows a device, known as a **P-box** (P stands for permutation), used to effect a transposition on an 8-bit input. If the 8 bits are designated from top to bottom as 01234567, the output of this particular P-box is 36071245. By appropriate internal wiring, a P-box can be made to perform any transposition and do it at practically the speed of light since no computation is involved, just signal propagation. This design follows Kerckhoff's principle: the attacker knows that the general method is permuting the bits. What he does not know is which bit goes where.

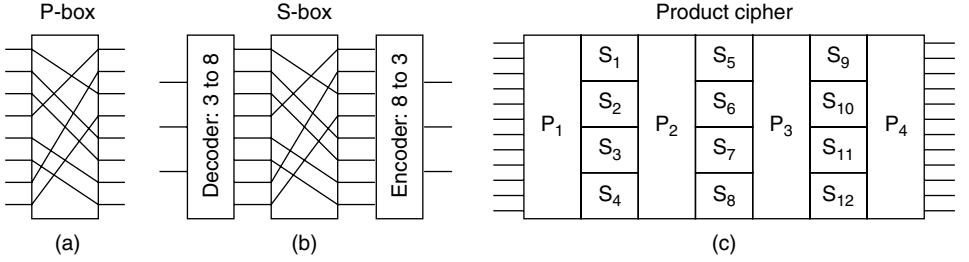


Figure 8-6. Basic elements of product ciphers. (a) P-box. (b) S-box. (c) Product.

Substitutions are performed by **S-boxes**, as shown in Fig. 8-6(b). In this example, a 3-bit plaintext is entered and a 3-bit ciphertext is output. The 3-bit input selects one of the eight lines exiting from the first stage and sets it to 1; all the other lines are 0. The second stage is a P-box. The third stage encodes the selected input line in binary again. With the wiring shown, if the eight octal numbers 01234567 were input one after another, the output sequence would be 24506713. In other words, 0 has been replaced by 2, 1 has been replaced by 4, etc. Again, by appropriate wiring of the P-box inside the S-box, any substitution can be accomplished. Furthermore, such a device can be built in hardware to achieve great speed, since encoders and decoders have only one or two (subnanosecond) gate delays and the propagation time across the P-box may well be less than 1 picosec.

The real power of these basic elements only becomes apparent when we cascade a whole series of boxes to form a **product cipher**, as shown in Fig. 8-6(c). In this example, 12 input lines are transposed (i.e., permuted) by the first stage (P_1). In the second stage, the input is broken up into four groups of 3 bits, each of which is substituted independently of the others (S_1 to S_4). This arrangement shows a method of approximating a larger S-box from multiple, smaller S-boxes. It is useful because small S-boxes are practical for a hardware implementation (e.g., an 8-bit S-box can be realized as a 256-entry lookup table), but large S-boxes become unwieldy to build (e.g., a 12-bit S-box would at a minimum need $2^{12} = 4096$ crossed wires in its middle stage). Although this method is less general, it is still powerful. By inclusion of a sufficiently large number of stages in the product cipher, the output can be made to be an exceedingly complicated function of the input.

Product ciphers that operate on k-bit inputs to produce k-bit outputs are very common. Typically, k is 64 to 256. A hardware implementation usually has at least 10 physical stages, instead of just 7 as in Fig. 8-6(c). A software implementation is programmed as a loop with at least eight iterations, each one performing S-box-type substitutions on subblocks of the 64- to 256-bit data block, followed by a permutation that mixes the outputs of the S-boxes. Often there is a special initial permutation and one at the end as well. In the literature, the iterations are called **rounds**.

8.2.1 DES—The Data Encryption Standard

In January 1977, the U.S. Government adopted a product cipher developed by IBM as its official standard for unclassified information. This cipher, **DES** (**Data Encryption Standard**), was widely adopted by the industry for use in security products. It is no longer secure in its original form, but in a modified form it is still useful. We will now explain how DES works.

An outline of DES is shown in Fig. 8-7(a). Plaintext is encrypted in blocks of 64 bits, yielding 64 bits of ciphertext. The algorithm, which is parameterized by a 56-bit key, has 19 distinct stages. The first stage is a key-independent transposition on the 64-bit plaintext. The last stage is the exact inverse of this transposition. The stage prior to the last one exchanges the leftmost 32 bits with the rightmost 32 bits. The remaining 16 stages are functionally identical but are parameterized by different functions of the key. The algorithm has been designed to allow decryption to be done with the same key as encryption, a property needed in any symmetric-key algorithm. The steps are just run in the reverse order.

The operation of one of these intermediate stages is illustrated in Fig. 8-7(b). Each stage takes two 32-bit inputs and produces two 32-bit outputs. The left output is simply a copy of the right input. The right output is the bitwise XOR of the left input and a function of the right input and the key for this stage, K_i. Pretty much all the complexity of the algorithm lies in this function.

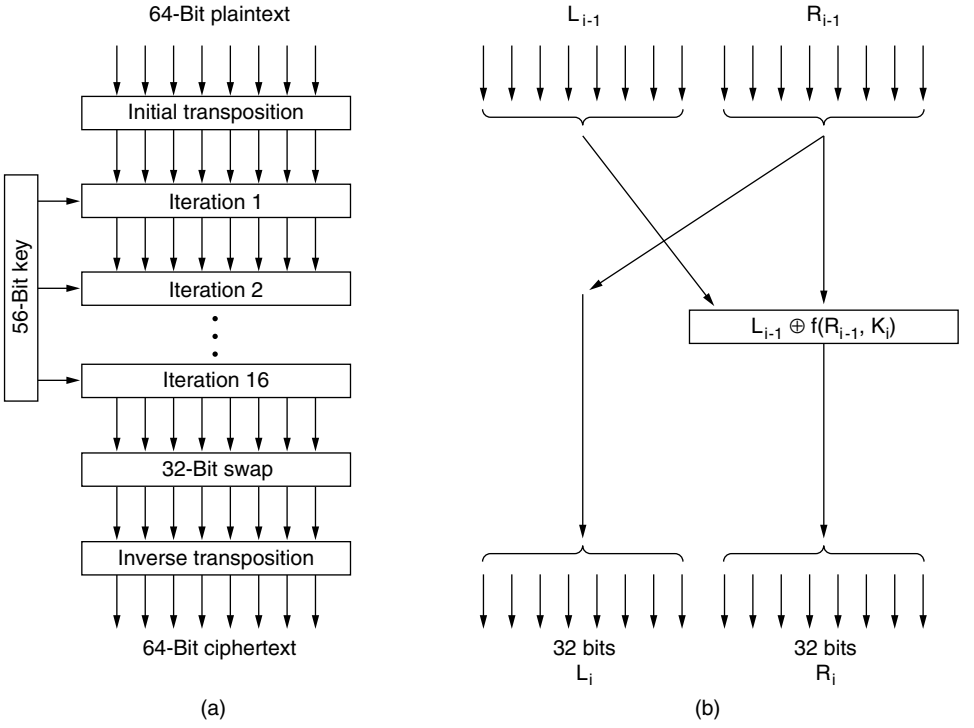


Figure 8-7. The Data Encryption Standard. (a) General outline. (b) Detail of one iteration. The circled + means exclusive OR.

The function consists of four steps, carried out in sequence. First, a 48-bit number, E, is constructed by expanding the 32-bit R_{i-1} according to a fixed transposition and duplication rule. Second, E and K_i are XORed together. This output is then partitioned into eight groups of 6 bits each, each of which is fed into a different S-box. Each of the 64 possible inputs to an S-box is mapped onto a 4-bit output. Finally, these 8×4 bits are passed through a P-box.

In each of the 16 iterations, a different key is used. Before the algorithm starts, a 56-bit transposition is applied to the key. Just before each iteration, the key is partitioned into two 28-bit units, each of which is rotated left by a number of bits dependent on the iteration number. K_i is derived from this rotated key by applying yet another 56-bit transposition to it. A different 48-bit subset of the 56 bits is extracted and permuted on each round.

A technique that is sometimes used to make DES stronger is called **whitening**. It consists of XORing a random 64-bit key with each plaintext block before feeding it into DES and then XORing a second 64-bit key with the resulting ciphertext before transmitting it. Whitening can easily be removed by running the

reverse operations (if the receiver has the two whitening keys). Since this technique effectively adds more bits to the key length, it makes an exhaustive search of the key space much more time consuming. Note that the same whitening key is used for each block (i.e., there is only one whitening key).

DES has been enveloped in controversy since the day it was launched. It was based on a cipher developed and patented by IBM, called Lucifer, except that IBM's cipher used a 128-bit key instead of a 56-bit key. When the U.S. Federal Government wanted to standardize on one cipher for unclassified use, it "invited" IBM to "discuss" the matter with NSA, the U.S. Government's code-breaking arm, which is the world's largest employer of mathematicians and cryptologists. NSA is so secret that an industry joke goes:

Q: What does NSA stand for?
A: No Such Agency.

Actually, NSA stands for National Security Agency.

After these discussions took place, IBM reduced the key from 128 bits to 56 bits and decided to keep secret the process by which DES was designed. Many people suspected that the key length was reduced to make sure that NSA could just break DES, but no organization with a smaller budget could. The point of the secret design was supposedly to hide a back door that could make it even easier for NSA to break DES. When an NSA employee discreetly told IEEE to cancel a planned conference on cryptography, that did not make people any more comfortable. NSA denied everything.

In 1977, two Stanford cryptography researchers, Diffie and Hellman (1977), designed a machine to break DES and estimated that it could be built for 20 million dollars. Given a small piece of plaintext and matched ciphertext, this machine could find the key by exhaustive search of the 2^{56}-entry key space in under 1 day. Nowadays, the game is up. Such a machine exists, is for sale, and costs less than $10,000 to make (Kumar et al., 2006).

Triple DES

As early as 1979, IBM realized that the DES key length was too short and devised a way to effectively increase it, using triple encryption (Tuchman, 1979). The method chosen, which has since been incorporated in International Standard 8732, is illustrated in Fig. 8-8. Here, two keys and three stages are used. In the first stage, the plaintext is encrypted using DES in the usual way with K_1. In the second stage, DES is run in decryption mode, using K_2 as the key. Finally, another DES encryption is done with K_1.

This design immediately gives rise to two questions. First, why are only two keys used, instead of three? Second, why is **EDE** (**Encrypt Decrypt Encrypt**) used, instead of **EEE** (**Encrypt Encrypt Encrypt**)? The reason that two keys are used is that even the most paranoid of cryptographers believe that 112 bits is

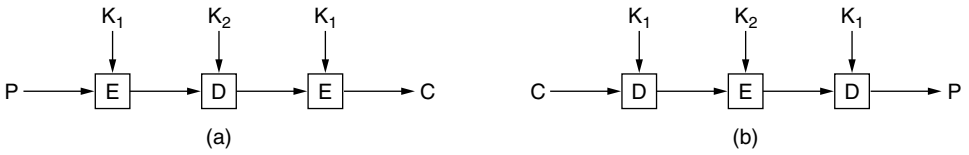


Figure 8-8. (a) Triple encryption using DES. (b) Decryption.

adequate for routine commercial applications for the time being. (And among cryptographers, paranoia is considered a feature, not a bug.) Going to 168 bits would just add the unnecessary overhead of managing and transporting another key for little real gain.

The reason for encrypting, decrypting, and then encrypting again is backward compatibility with existing single-key DES systems. Both the encryption and decryption functions are mappings between sets of 64-bit numbers. From a cryptographic point of view, the two mappings are equally strong. By using EDE, however, instead of EEE, a computer using triple encryption can speak to one using single encryption by just setting $K_1 = K_2$. This property allows triple encryption to be phased in gradually, something of no concern to academic cryptographers but of considerable importance to IBM and its customers.

8.2.2 AES—The Advanced Encryption Standard

As DES began approaching the end of its useful life, even with triple DES, **NIST** (**National Institute of Standards and Technology**), the agency of the U.S. Dept. of Commerce charged with approving standards for the U.S. Federal Government, decided that the government needed a new cryptographic standard for unclassified use. NIST was keenly aware of all the controversy surrounding DES and well knew that if it just announced a new standard, everyone knowing anything about cryptography would automatically assume that NSA had built a back door into it so NSA could read everything encrypted with it. Under these conditions, probably no one would use the standard and it would have died quietly.

So, NIST took a surprisingly different approach for a government bureaucracy: it sponsored a cryptographic bake-off (contest). In January 1997, researchers from all over the world were invited to submit proposals for a new standard, to be called **AES** (**Advanced Encryption Standard**). The bake-off rules were:

1. The algorithm must be a symmetric block cipher.
2. The full design must be public.
3. Key lengths of 128, 192, and 256 bits must be supported.

4. Both software and hardware implementations must be possible.
5. The algorithm must be public or licensed on nondiscriminatory terms.

Fifteen serious proposals were made, and public conferences were organized in which they were presented and attendees were actively encouraged to find flaws in all of them. In August 1998, NIST selected five finalists, primarily on the basis of their security, efficiency, simplicity, flexibility, and memory requirements (important for embedded systems). More conferences were held and more potshots taken.

In October 2000, NIST announced that it had selected Rijndael, by Joan Daemen and Vincent Rijmen. The name Rijndael, pronounced Rhine-doll (more or less), is derived from the last names of the authors: Rijmen + Daemen. In November 2001, Rijndael became the AES U.S. Government standard, published as FIPS (Federal Information Processing Standard) 197. Due to the extraordinary openness of the competition, the technical properties of Rijndael, and the fact that the winning team consisted of two young Belgian cryptographers (who were unlikely to have built in a back door just to please NSA), Rijndael has become the world's dominant cryptographic cipher. AES encryption and decryption is now part of the instruction set for some microprocessors (e.g., Intel).

Rijndael supports key lengths and block sizes from 128 bits to 256 bits in steps of 32 bits. The key length and block length may be chosen independently. However, AES specifies that the block size must be 128 bits and the key length must be 128, 192, or 256 bits. It is doubtful that anyone will ever use 192-bit keys, so de facto, AES has two variants: a 128-bit block with a 128-bit key and a 128-bit block with a 256-bit key.

In our treatment of the algorithm, we will examine only the 128/128 case because this is likely to become the commercial norm. A 128-bit key gives a key space of $2^{128} \approx 3 \times 10^{38}$ keys. Even if NSA manages to build a machine with 1 billion parallel processors, each being able to evaluate one key per picosecond, it would take such a machine about 10^{10} years to search the key space. By then the sun will have burned out, so the folks then present will have to read the results by candlelight.

Rijndael

From a mathematical perspective, Rijndael is based on Galois field theory, which gives it some provable security properties. However, it can also be viewed as C code, without getting into the mathematics.

Like DES, Rijndael uses substitution and permutations, and it also uses multiple rounds. The number of rounds depends on the key size and block size, being 10 for 128-bit keys with 128-bit blocks and moving up to 14 for the largest key or the largest block. However, unlike DES, all operations involve entire bytes, to

allow for efficient implementations in both hardware and software. An outline of the code is given in Fig. 8-9. Note that this code is for the purpose of illustration. Good implementations of security code will follow additional practices, such as zeroing out sensitive memory after it has been used. See, for example, Ferguson et al. (2010).

```
#define LENGTH 16                                   /* # bytes in data block or key */
#define NROWS 4                                     /* number of rows in state */
#define NCOLS 4                                     /* number of columns in state */
#define ROUNDS 10                                   /* number of iterations */
typedef unsigned char byte;                         /* unsigned 8-bit integer */

rijndael(byte plaintext[LENGTH], byte ciphertext[LENGTH], byte key[LENGTH])
{
  int r;                                            /* loop index */
  byte state[NROWS][NCOLS];                         /* current state */
  struct {byte k[NROWS][NCOLS];} rk[ROUNDS + 1];    /* round keys */

  expand_key(key, rk);                              /* construct the round keys */
  copy_plaintext_to_state(state, plaintext);        /* init current state */
  xor_roundkey_into_state(state, rk[0]);            /* XOR key into state */

  for (r = 1; r <= ROUNDS; r++) {
      substitute(state);                            /* apply S-box to each byte */
      rotate_rows(state);                           /* rotate row i by i bytes */
      if (r < ROUNDS) mix_columns(state);           /* mix function */
      xor_roundkey_into_state(state, rk[r]);        /* XOR key into state */
  }
  copy_state_to_ciphertext(ciphertext, state);      /* return result */
}
```

Figure 8-9. An outline of Rijndael in C.

The function *rijndael* has three parameters. They are: *plaintext*, an array of 16 bytes containing the input data; *ciphertext*, an array of 16 bytes where the enciphered output will be returned; and *key*, the 16-byte key. During the calculation, the current state of the data is maintained in a byte array, *state*, whose size is $NROWS \times NCOLS$. For 128-bit blocks, this array is 4×4 bytes. With 16 bytes, the full 128-bit data block can be stored.

The *state* array is initialized to the plaintext and modified by every step in the computation. In some steps, byte-for-byte substitution is performed. In others, the bytes are permuted within the array. Other transformations are also used. At the end, the contents of the *state* are returned as the ciphertext.

The code starts out by expanding the key into 11 arrays of the same size as the state. They are stored in *rk*, which is an array of structs, each containing a state array. One of these will be used at the start of the calculation and the other 10 will be used during the 10 rounds, one per round. The calculation of the round

keys from the encryption key is too complicated for us to get into here. Suffice it to say that the round keys are produced by repeated rotation and XORing of various groups of key bits. For all the details, see Daemen and Rijmen (2002).

The next step is to copy the plaintext into the *state* array so it can be processed during the rounds. It is copied in column order, with the first 4 bytes going into column 0, the next 4 bytes going into column 1, and so on. Both the columns and the rows are numbered starting at 0, although the rounds are numbered starting at 1. This initial setup of the 12 byte arrays of size 4×4 is illustrated in Fig. 8-10.

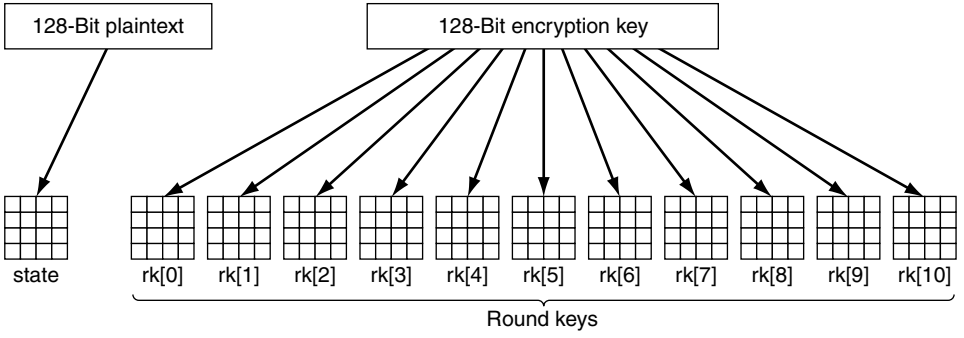


Figure 8-10. Creating the *state* and *rk* arrays.

There is one more step before the main computation begins: *rk*[0] is XORed into *state*, byte for byte. In other words, each of the 16 bytes in *state* is replaced by the XOR of itself and the corresponding byte in *rk*[0].

Now it is time for the main attraction. The loop executes 10 iterations, one per round, transforming *state* on each iteration. The contents of each round is produced in four steps. Step 1 does a byte-for-byte substitution on *state*. Each byte in turn is used as an index into an S-box to replace its value by the contents of that S-box entry. This step is a straight monoalphabetic substitution cipher. Unlike DES, which has multiple S-boxes, Rijndael has only one S-box.

Step 2 rotates each of the four rows to the left. Row 0 is rotated 0 bytes (i.e., not changed), row 1 is rotated 1 byte, row 2 is rotated 2 bytes, and row 3 is rotated 3 bytes. This step diffuses the contents of the current data around the block, analogous to the permutations of Fig. 8-6.

Step 3 mixes up each column independently of the other ones. The mixing is done using matrix multiplication in which the new column is the product of the old column and a constant matrix, with the multiplication done using the finite Galois field, $GF(2^8)$. Although this may sound complicated, an algorithm exists that allows each element of the new column to be computed using two table lookups and three XORs (Daemen and Rijmen, 2002, Appendix E).

Finally, step 4 XORs the key for this round into the *state* array for use in the next round.

Since every step is reversible, decryption can be done just by running the algorithm backward. However, there is also a trick available in which decryption can be done by running the encryption algorithm using different tables.

The algorithm has been designed not only for great security, but also for great speed. A good software implementation on a 2-GHz machine should be able to achieve an encryption rate of 700 Mbps, which is fast enough to encrypt over 100 MPEG-2 videos in real time. Hardware implementations are faster still.

8.2.3 Cipher Modes

Despite all this complexity, AES (or DES, or any block cipher for that matter) is basically a monoalphabetic substitution cipher using big characters (128-bit characters for AES and 64-bit characters for DES). Whenever the same plaintext block goes in the front end, the same ciphertext block comes out the back end. If you encrypt the plaintext *abcdefgh* 100 times with the same DES key, you get the same ciphertext 100 times. An intruder can exploit this property to help subvert the cipher.

Electronic Code Book Mode

To see how this monoalphabetic substitution cipher property can be used to partially defeat the cipher, we will use (triple) DES because it is easier to depict 64-bit blocks than 128-bit blocks, but AES has exactly the same problem. The straightforward way to use DES to encrypt a long piece of plaintext is to break it up into consecutive 8-byte (64-bit) blocks and encrypt them one after another with the same key. The last piece of plaintext is padded out to 64 bits, if need be. This technique is known as **ECB mode** (**Electronic Code Book mode**) in analogy with old-fashioned code books where each plaintext word was listed, followed by its ciphertext (usually a five-digit decimal number).

In Fig. 8-11, we have the start of a computer file listing the annual bonuses a company has decided to award to its employees. This file consists of consecutive 32-byte records, one per employee, in the format shown: 16 bytes for the name, 8 bytes for the position, and 8 bytes for the bonus. Each of the sixteen 8-byte blocks (numbered from 0 to 15) is encrypted by (triple) DES.

Leslie just had a fight with the boss and is not expecting much of a bonus. Kim, in contrast, is the boss' favorite, and everyone knows this. Leslie can get access to the file after it is encrypted but before it is sent to the bank. Can Leslie rectify this unfair situation, given only the encrypted file?

No problem at all. All Leslie has to do is make a copy of the 12th ciphertext block (which contains Kim's bonus) and use it to replace the fourth ciphertext block (which contains Leslie's bonus). Even without knowing what the 12th

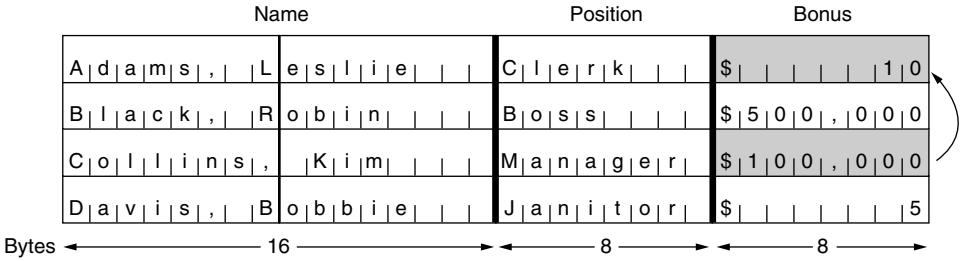


Figure 8-11. The plaintext of a file encrypted as 16 DES blocks.

block says, Leslie can expect to have a much merrier Christmas this year. (Copying the eighth ciphertext block is also a possibility, but is more likely to be detected; besides, Leslie is not a greedy person.)

Cipher Block Chaining Mode

To thwart this type of attack, all block ciphers can be chained in various ways so that replacing a block the way Leslie did will cause the plaintext decrypted starting at the replaced block to be garbage. One way of chaining is **cipher block chaining**. In this method, shown in Fig. 8-12, each plaintext block is XORed with the previous ciphertext block before being encrypted. Consequently, the same plaintext block no longer maps onto the same ciphertext block, and the encryption is no longer a big monoalphabetic substitution cipher. The first block is XORed with a randomly chosen **IV** (**Initialization Vector**), which is transmitted (in plaintext) along with the ciphertext.

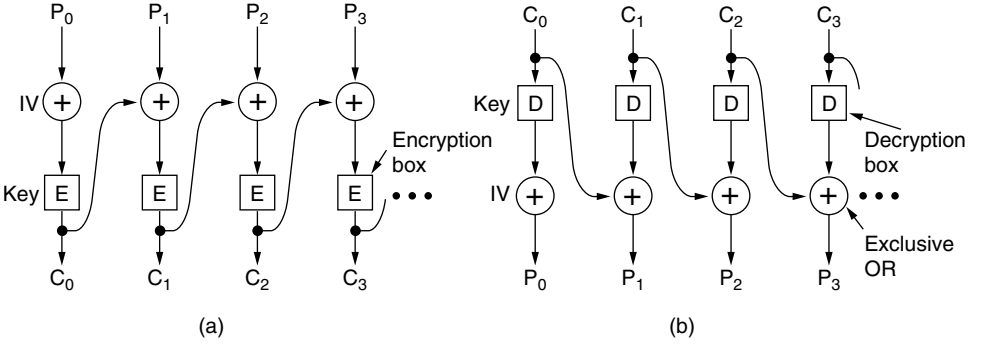


Figure 8-12. Cipher block chaining. (a) Encryption. (b) Decryption.

We can see how cipher block chaining mode works by examining the example of Fig. 8-12. We start out by computing $C_0 = E(P_0 \text{ XOR } IV)$. Then we compute $C_1 = E(P_1 \text{ XOR } C_0)$, and so on. Decryption also uses XOR to reverse the process, with $P_0 = IV \text{ XOR } D(C_0)$, and so on. Note that the encryption of block i is a

function of all the plaintext in blocks 0 through $i - 1$, so the same plaintext generates different ciphertext depending on where it occurs. A transformation of the type Leslie made will result in nonsense for two blocks starting at Leslie's bonus field. To an astute security officer, this peculiarity might suggest where to start the ensuing investigation.

Cipher block chaining also has the advantage that the same plaintext block will not result in the same ciphertext block, making cryptanalysis more difficult. In fact, this is the main reason it is used.

Cipher Feedback Mode

However, cipher block chaining has the disadvantage of requiring an entire 64-bit block to arrive before decryption can begin. For byte-by-byte encryption, **cipher feedback mode** using (triple) DES is used, as shown in Fig. 8-13. For AES, the idea is exactly the same, only a 128-bit shift register is used. In this figure, the state of the encryption machine is shown after bytes 0 through 9 have been encrypted and sent. When plaintext byte 10 arrives, as illustrated in Fig. 8-13(a), the DES algorithm operates on the 64-bit shift register to generate a 64-bit ciphertext. The leftmost byte of that ciphertext is extracted and XORed with P_{10}. That byte is transmitted on the transmission line. In addition, the shift register is shifted left 8 bits, causing C_2 to fall off the left end, and C_{10} is inserted in the position just vacated at the right end by C_9.

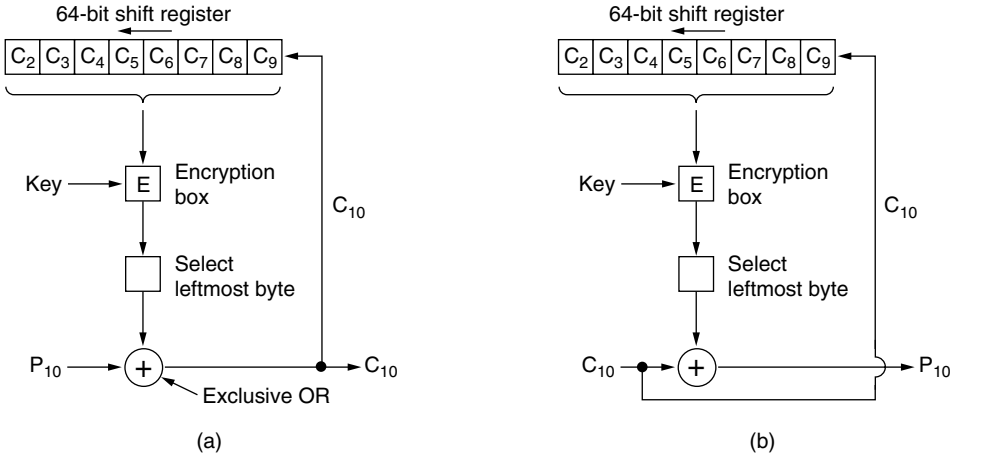


Figure 8-13. Cipher feedback mode. (a) Encryption. (b) Decryption.

Note that the contents of the shift register depend on the entire previous history of the plaintext, so a pattern that repeats multiple times in the plaintext will be encrypted differently each time in the ciphertext. As with cipher block chaining, an initialization vector is needed to start the ball rolling.

Decryption with cipher feedback mode works the same way as encryption. In particular, the content of the shift register is *encrypted*, not *decrypted*, so the selected byte that is XORed with C_{10} to get P_{10} is the same one that was XORed with P_{10} to generate C_{10} in the first place. As long as the two shift registers remain identical, decryption works correctly. This is illustrated in Fig. 8-13(b).

A problem with cipher feedback mode is that if one bit of the ciphertext is accidentally inverted during transmission, the 8 bytes that are decrypted while the bad byte is in the shift register will be corrupted. Once the bad byte is pushed out of the shift register, correct plaintext will once again be generated. Thus, the effects of a single inverted bit are relatively localized and do not ruin the rest of the message, but they do ruin as many bits as the shift register is wide.

Stream Cipher Mode

Nevertheless, applications exist in which having a 1-bit transmission error mess up 64 bits of plaintext is too large an effect. For these applications, a fourth option, **stream cipher mode**, exists. It works by encrypting an initialization vector, using a key to get an output block. The output block is then encrypted, using the key to get a second output block. This block is then encrypted to get a third block, and so on. The (arbitrarily large) sequence of output blocks, called the **keystream**, is treated like a one-time pad and XORed with the plaintext to get the ciphertext, as shown in Fig. 8-14(a). Note that the IV is used only on the first step. After that, the output is encrypted. Also note that the keystream is independent of the data, so it can be computed in advance, if need be, and is completely insensitive to transmission errors. Decryption is shown in Fig. 8-14(b).

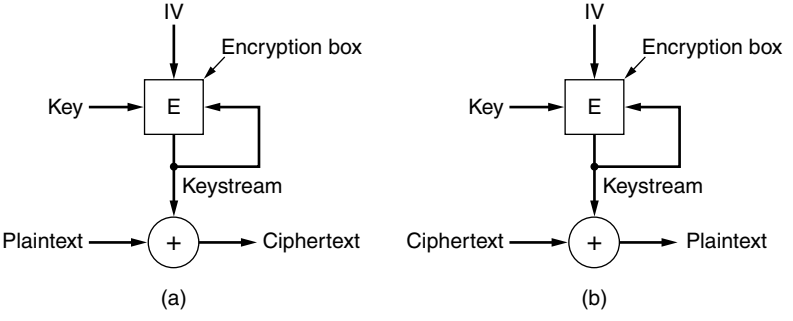


Figure 8-14. A stream cipher. (a) Encryption. (b) Decryption.

Decryption occurs by generating the same keystream at the receiving side. Since the keystream depends only on the IV and the key, it is not affected by transmission errors in the ciphertext. Thus, a 1-bit error in the transmitted ciphertext generates only a 1-bit error in the decrypted plaintext.

It is essential never to use the same (key, IV) pair twice with a stream cipher because doing so will generate the same keystream each time. Using the same keystream twice exposes the ciphertext to a **keystream reuse attack**. Imagine that the plaintext block, P_0, is encrypted with the keystream to get P_0 XOR K_0. Later, a second plaintext block, Q_0, is encrypted with the same keystream to get Q_0 XOR K_0. An intruder who captures both of these ciphertext blocks can simply XOR them together to get P_0 XOR Q_0, which eliminates the key. The intruder now has the XOR of the two plaintext blocks. If one of them is known or can be guessed, the other can also be found. In any event, the XOR of two plaintext streams can be attacked by using statistical properties of the message. For example, for English text, the most common character in the stream will probably be the XOR of two spaces, followed by the XOR of space and the letter "e", etc. In short, equipped with the XOR of two plaintexts, the cryptanalyst has an excellent chance of deducing both of them.

Counter Mode

One problem that all the modes except electronic code book mode have is that random access to encrypted data is impossible. For example, suppose a file is transmitted over a network and then stored on disk in encrypted form. This might be a reasonable way to operate if the receiving computer is a notebook computer that might be stolen. Storing all critical files in encrypted form greatly reduces the damage due to secret information leaking out in the event that the computer falls into the wrong hands.

However, disk files are often accessed in nonsequential order, especially files in databases. With a file encrypted using cipher block chaining, accessing a random block requires first decrypting all the blocks ahead of it, an expensive proposition. For this reason, yet another mode has been invented: **counter mode**, as illustrated in Fig. 8-15. Here, the plaintext is not encrypted directly. Instead, the initialization vector plus a constant is encrypted, and the resulting ciphertext is XORed with the plaintext. By stepping the initialization vector by 1 for each new block, it is easy to decrypt a block anywhere in the file without first having to decrypt all of its predecessors.

Although counter mode is useful, it has a weakness that is worth pointing out. Suppose that the same key, K, is used again in the future (with a different plaintext but the same IV) and an attacker acquires all the ciphertext from both runs. The keystreams are the same in both cases, exposing the cipher to a keystream reuse attack of the same kind we saw with stream ciphers. All the cryptanalyst has to do is XOR the two ciphertexts together to eliminate all the cryptographic protection and just get the XOR of the plaintexts. This weakness does not mean counter mode is a bad idea. It just means that both keys and initialization vectors should be chosen independently and at random. Even if the same key is accidentally used twice, if the IV is different each time, the plaintext is safe.

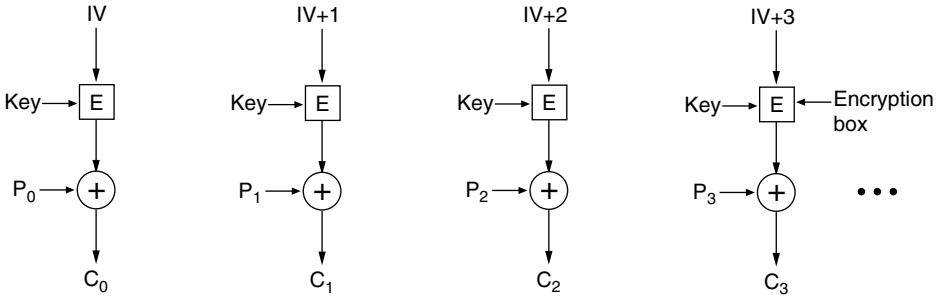


Figure 8-15. Encryption using counter mode.

8.2.4 Other Ciphers

AES (Rijndael) and DES are the best-known symmetric-key cryptographic algorithms, and the standard industry choices, if only for liability reasons. (No one will blame you if you use AES in your product and AES is cracked, but they will certainly blame you if you use a nonstandard cipher and it is later broken.) However, it is worth mentioning that numerous other symmetric-key ciphers have been devised. Some of these are embedded inside various products. A few of the more common ones are listed in Fig. 8-16. It is possible to use combinations of these ciphers, for example, AES over Twofish, so that both ciphers need to be broken to recover the data.

Cipher	Author	Key length	Comments
DES	IBM	56 bits	Too weak to use now
RC4	Ronald Rivest	1–2048 bits	Caution: some keys are weak
RC5	Ronald Rivest	128–256 bits	Good, but patented
AES (Rijndael)	Daemen and Rijmen	128–256 bits	Best choice
Serpent	Anderson, Biham, Knudsen	128–256 bits	Very strong
Triple DES	IBM	168 bits	Good, but getting old
Twofish	Bruce Schneier	128–256 bits	Very strong; widely used

Figure 8-16. Some common symmetric-key cryptographic algorithms.

8.2.5 Cryptanalysis

Before leaving the subject of symmetric-key cryptography, it is worth at least mentioning four developments in cryptanalysis. The first development is **differential cryptanalysis** (Biham and Shamir, 1997). This technique can be used

to attack any block cipher. It works by beginning with a pair of plaintext blocks differing in only a small number of bits and watching carefully what happens on each internal iteration as the encryption proceeds. In many cases, some bit patterns are more common than others, which can lead to probabilistic attacks.

The second development worth noting is **linear cryptanalysis** (Matsui, 1994). It can break DES with only 2^{43} known plaintexts. It works by XORing certain bits in the plaintext and ciphertext together and examining the result. When done repeatedly, half the bits should be 0s and half should be 1s. Often, however, ciphers introduce a bias in one direction or the other, and this bias, however small, can be exploited to reduce the work factor. For the details, see Matsui's paper.

The third development is using analysis of electrical power consumption to find secret keys. Computers typically use around 3 volts to represent a 1 bit and 0 volts to represent a 0 bit. Thus, processing a 1 takes more electrical energy than processing a 0. If a cryptographic algorithm consists of a loop in which the key bits are processed in order, an attacker who replaces the main *n*-GHz clock with a slow (e.g., 100-Hz) clock and puts alligator clips on the CPU's power and ground pins can precisely monitor the power consumed by each machine instruction. From this data, deducing the key is surprisingly easy. This kind of cryptanalysis can be defeated only by carefully coding the algorithm in assembly language to make sure power consumption is independent of the key and also independent of all the individual round keys.

The fourth development is timing analysis. Cryptographic algorithms are full of if statements that test bits in the round keys. If the then and else parts take different amounts of time, by slowing down the clock and seeing how long various steps take, it may also be possible to deduce the round keys. Once all the round keys are known, the original key can usually be computed. Power and timing analysis can also be employed simultaneously to make the job easier. While power and timing analysis may seem exotic, in reality they are powerful techniques that can break any cipher not specifically designed to resist them.

8.3 PUBLIC-KEY ALGORITHMS

Historically, distributing the keys has always been the weakest link in most cryptosystems. No matter how strong a cryptosystem was, if an intruder could steal the key, the system was worthless. Cryptologists always took for granted that the encryption key and decryption key were the same (or easily derived from one another). But the key had to be distributed to all users of the system. Thus, it seemed as if there was an inherent problem. Keys had to be protected from theft, but they also had to be distributed, so they could not be locked in a bank vault.

In 1976, two researchers at Stanford University, Diffie and Hellman (1976), proposed a radically new kind of cryptosystem, one in which the encryption and decryption keys were so different that the decryption key could not feasibly be

derived from the encryption key. In their proposal, the (keyed) encryption algorithm, E, and the (keyed) decryption algorithm, D, had to meet three requirements. These requirements can be stated simply as follows:

1. $D(E(P)) = P$.
2. It is exceedingly difficult to deduce D from E.
3. E cannot be broken by a chosen plaintext attack.

The first requirement says that if we apply D to an encrypted message, $E(P)$, we get the original plaintext message, P, back. Without this property, the legitimate receiver could not decrypt the ciphertext. The second requirement speaks for itself. The third requirement is needed because, as we shall see in a moment, intruders may experiment with the algorithm to their hearts' content. Under these conditions, there is no reason that the encryption key cannot be made public.

The method works like this. A person, say, Alice, who wants to receive secret messages, first devises two algorithms meeting the above requirements. The encryption algorithm and Alice's key are then made public, hence the name **public-key cryptography**. Alice might put her public key on her home page on the Web, for example. We will use the notation E_A to mean the encryption algorithm parameterized by Alice's public key. Similarly, the (secret) decryption algorithm parameterized by Alice's private key is D_A. Bob does the same thing, publicizing E_B but keeping D_B secret.

Now let us see if we can solve the problem of establishing a secure channel between Alice and Bob, who have never had any previous contact. Both Alice's encryption key, E_A, and Bob's encryption key, E_B, are assumed to be in publicly readable files. Now Alice takes her first message, P, computes $E_B(P)$, and sends it to Bob. Bob then decrypts it by applying his secret key D_B [i.e., he computes $D_B(E_B(P)) = P$]. No one else can read the encrypted message, $E_B(P)$, because the encryption system is assumed to be strong and because it is too difficult to derive D_B from the publicly known E_B. To send a reply, R, Bob transmits $E_A(R)$. Alice and Bob can now communicate securely.

A note on terminology is perhaps useful here. Public-key cryptography requires each user to have two keys: a public key, used by the entire world for encrypting messages to be sent to that user, and a private key, which the user needs for decrypting messages. We will consistently refer to these keys as the *public* and *private* keys, respectively, and distinguish them from the *secret* keys used for conventional symmetric-key cryptography.

8.3.1 RSA

The only catch is that we need to find algorithms that indeed satisfy all three requirements. Due to the potential advantages of public-key cryptography, many researchers are hard at work, and some algorithms have already been published.

One good method was discovered by a group at M.I.T. (Rivest et al., 1978). It is known by the initials of the three discoverers (Rivest, Shamir, Adleman): **RSA**. It has survived all attempts to break it for more than 30 years and is considered very strong. Much practical security is based on it. For this reason, Rivest, Shamir, and Adleman were given the 2002 ACM Turing Award. Its major disadvantage is that it requires keys of at least 1024 bits for good security (versus 128 bits for symmetric-key algorithms), which makes it quite slow.

The RSA method is based on some principles from number theory. We will now summarize how to use the method; for details, consult the paper.

1. Choose two large primes, p and q (typically 1024 bits).
2. Compute $n = p \times q$ and $z = (p - 1) \times (q - 1)$.
3. Choose a number relatively prime to z and call it d.
4. Find e such that $e \times d = 1 \ mod \ z$.

With these parameters computed in advance, we are ready to begin encryption. Divide the plaintext (regarded as a bit string) into blocks, so that each plaintext message, P, falls in the interval $0 \le P < n$. Do that by grouping the plaintext into blocks of k bits, where k is the largest integer for which $2^k < n$ is true.

To encrypt a message, P, compute $C = P^e \pmod{n}$. To decrypt C, compute $P = C^d \pmod{n}$. It can be proven that for all P in the specified range, the encryption and decryption functions are inverses. To perform the encryption, you need e and n. To perform the decryption, you need d and n. Therefore, the public key consists of the pair (e, n) and the private key consists of (d, n).

The security of the method is based on the difficulty of factoring large numbers. If the cryptanalyst could factor the (publicly known) n, he could then find p and q, and from these z. Equipped with knowledge of z and e, d can be found using Euclid's algorithm. Fortunately, mathematicians have been trying to factor large numbers for at least 300 years, and the accumulated evidence suggests that it is an exceedingly difficult problem.

According to Rivest and colleagues, factoring a 500-digit number would require 10^{25} years using brute force. In both cases, they assumed the best known algorithm and a computer with a 1-μsec instruction time. With a million chips running in parallel, each with an instruction time of 1 nsec, it would still take 10^{16} years. Even if computers continue to get faster by an order of magnitude per decade, it will be many years before factoring a 500-digit number becomes feasible, at which time our descendants can simply choose p and q still larger.

A trivial pedagogical example of how the RSA algorithm works is given in Fig. 8-17. For this example, we have chosen $p = 3$ and $q = 11$, giving $n = 33$ and $z = 20$. A suitable value for d is $d = 7$, since 7 and 20 have no common factors. With these choices, e can be found by solving the equation $7e = 1 \pmod{20}$, which yields $e = 3$. The ciphertext, C, corresponding to a plaintext message, P, is

given by $C = P^3 \pmod{33}$. The ciphertext is decrypted by the receiver by making use of the rule $P = C^7 \pmod{33}$. The figure shows the encryption of the plaintext "SUZANNE" as an example.

Plaintext (P)			Ciphertext (C)		After decryption	
Symbolic	Numeric	P^3	$P^3 \pmod{33}$	C^7	$C^7 \pmod{33}$	Symbolic
S	19	6859	28	13492928512	19	S
U	21	9261	21	1801088541	21	U
Z	26	17576	20	1280000000	26	Z
A	01	1	1	1	01	A
N	14	2744	5	78125	14	N
N	14	2744	5	78125	14	N
E	05	125	26	8031810176	05	E
Sender's computation				Receiver's computation		

Figure 8-17. An example of the RSA algorithm.

Because the primes chosen for this example are so small, P must be less than 33, so each plaintext block can contain only a single character. The result is a monoalphabetic substitution cipher, not very impressive. If instead we had chosen p and $q \approx 2^{512}$, we would have $n \approx 2^{1024}$, so each block could be up to 1024 bits or 128 eight-bit characters, versus 8 characters for DES and 16 characters for AES.

It should be pointed out that using RSA as we have described is similar to using a symmetric algorithm in ECB mode—the same input block gives the same output block. Therefore, some form of chaining is needed for data encryption. However, in practice, most RSA-based systems use public-key cryptography primarily for distributing one-time session keys for use with some symmetric-key algorithm such as AES or triple DES. RSA is too slow for actually encrypting large volumes of data but is widely used for key distribution.

8.3.2 Other Public-Key Algorithms

Although RSA is widely used, it is by no means the only public-key algorithm known. The first public-key algorithm was the knapsack algorithm (Merkle and Hellman, 1978). The idea here is that someone owns a large number of objects, each with a different weight. The owner encodes the message by secretly selecting a subset of the objects and placing them in the knapsack. The total weight of the objects in the knapsack is made public, as is the list of all possible objects and their corresponding weights. The list of objects in the knapsack is kept secret. With certain additional restrictions, the problem of figuring out a possible list of objects with the given weight was thought to be computationally infeasible and formed the basis of the public-key algorithm.

The algorithm's inventor, Ralph Merkle, was quite sure that this algorithm could not be broken, so he offered a $100 reward to anyone who could break it. Adi Shamir (the "S" in RSA) promptly broke it and collected the reward. Undeterred, Merkle strengthened the algorithm and offered a $1000 reward to anyone who could break the new one. Ronald Rivest (the "R" in RSA) promptly broke the new one and collected the reward. Merkle did not dare offer $10,000 for the next version, so "A" (Leonard Adleman) was out of luck. Nevertheless, the knapsack algorithm is not considered secure and is not used in practice any more.

Other public-key schemes are based on the difficulty of computing discrete logarithms. Algorithms that use this principle have been invented by El Gamal (1985) and Schnorr (1991).

A few other schemes exist, such as those based on elliptic curves (Menezes and Vanstone, 1993), but the two major categories are those based on the difficulty of factoring large numbers and computing discrete logarithms modulo a large prime. These problems are thought to be genuinely difficult to solve—mathematicians have been working on them for many years without any great breakthroughs.

8.4 DIGITAL SIGNATURES

The authenticity of many legal, financial, and other documents is determined by the presence or absence of an authorized handwritten signature. And photocopies do not count. For computerized message systems to replace the physical transport of paper-and-ink documents, a method must be found to allow documents to be signed in an unforgeable way.

The problem of devising a replacement for handwritten signatures is a difficult one. Basically, what is needed is a system by which one party can send a signed message to another party in such a way that the following conditions hold:

1. The receiver can verify the claimed identity of the sender.
2. The sender cannot later repudiate the contents of the message.
3. The receiver cannot possibly have concocted the message himself.

The first requirement is needed, for example, in financial systems. When a customer's computer orders a bank's computer to buy a ton of gold, the bank's computer needs to be able to make sure that the computer giving the order really belongs to the customer whose account is to be debited. In other words, the bank has to authenticate the customer (and the customer has to authenticate the bank).

The second requirement is needed to protect the bank against fraud. Suppose that the bank buys the ton of gold, and immediately thereafter the price of gold

drops sharply. A dishonest customer might then proceed to sue the bank, claiming that he never issued any order to buy gold. When the bank produces the message in court, the customer may deny having sent it. The property that no party to a contract can later deny having signed it is called **nonrepudiation**. The digital signature schemes that we will now study help provide it.

The third requirement is needed to protect the customer in the event that the price of gold shoots up and the bank tries to construct a signed message in which the customer asked for one bar of gold instead of one ton. In this fraud scenario, the bank just keeps the rest of the gold for itself.

8.4.1 Symmetric-Key Signatures

One approach to digital signatures is to have a central authority that knows everything and whom everyone trusts, say, Big Brother (*BB*). Each user then chooses a secret key and carries it by hand to *BB*'s office. Thus, only Alice and *BB* know Alice's secret key, K_A, and so on.

When Alice wants to send a signed plaintext message, *P*, to her banker, Bob, she generates $K_A(B, R_A, t, P)$, where *B* is Bob's identity, R_A is a random number chosen by Alice, *t* is a timestamp to ensure freshness, and $K_A(B, R_A, t, P)$ is the message encrypted with her key, K_A. Then she sends it as depicted in Fig. 8-18. *BB* sees that the message is from Alice, decrypts it, and sends a message to Bob as shown. The message to Bob contains the plaintext of Alice's message and also the signed message $K_{BB}(A, t, P)$. Bob now carries out Alice's request.

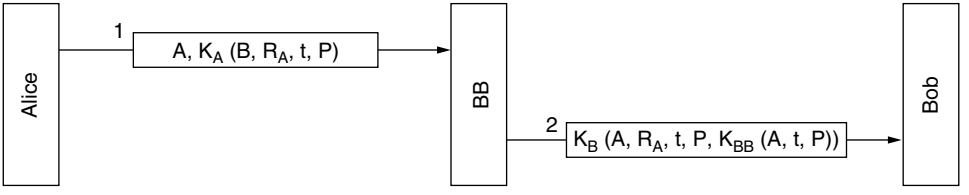


Figure 8-18. Digital signatures with Big Brother.

What happens if Alice later denies sending the message? Step 1 is that everyone sues everyone (at least, in the United States). Finally, when the case comes to court and Alice vigorously denies sending Bob the disputed message, the judge will ask Bob how he can be sure that the disputed message came from Alice and not from Trudy. Bob first points out that *BB* will not accept a message from Alice unless it is encrypted with K_A, so there is no possibility of Trudy sending *BB* a false message from Alice without BB detecting it immediately.

Bob then dramatically produces Exhibit A: $K_{BB}(A, t, P)$. Bob says that this is a message signed by *BB* that proves Alice sent *P* to Bob. The judge then asks *BB* (whom everyone trusts) to decrypt Exhibit A. When *BB* testifies that Bob is telling the truth, the judge decides in favor of Bob. Case dismissed.

One potential problem with the signature protocol of Fig. 8-18 is Trudy replaying either message. To minimize this problem, timestamps are used throughout. Furthermore, Bob can check all recent messages to see if R_A was used in any of them. If so, the message is discarded as a replay. Note that based on the timestamp, Bob will reject very old messages. To guard against instant replay attacks, Bob just checks the R_A of every incoming message to see if such a message has been received from Alice in the past hour. If not, Bob can safely assume this is a new request.

8.4.2 Public-Key Signatures

A structural problem with using symmetric-key cryptography for digital signatures is that everyone has to agree to trust Big Brother. Furthermore, Big Brother gets to read all signed messages. The most logical candidates for running the Big Brother server are the government, the banks, the accountants, and the lawyers. Unfortunately, none of these inspire total confidence in all citizens. Hence, it would be nice if signing documents did not require a trusted authority.

Fortunately, public-key cryptography can make an important contribution in this area. Let us assume that the public-key encryption and decryption algorithms have the property that $E(D(P)) = P$, in addition, of course, to the usual property that $D(E(P)) = P$. (RSA has this property, so the assumption is not unreasonable.) Assuming that this is the case, Alice can send a signed plaintext message, P, to Bob by transmitting $E_B(D_A(P))$. Note carefully that Alice knows her own (private) key, D_A, as well as Bob's public key, E_B, so constructing this message is something Alice can do.

When Bob receives the message, he transforms it using his private key, as usual, yielding $D_A(P)$, as shown in Fig. 8-19. He stores this text in a safe place and then applies E_A to get the original plaintext.

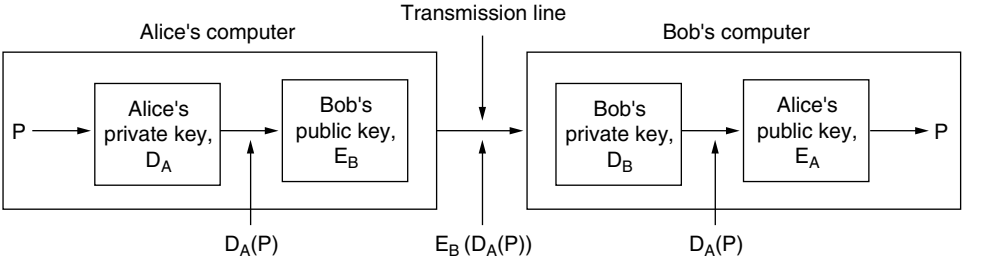


Figure 8-19. Digital signatures using public-key cryptography.

To see how the signature property works, suppose that Alice subsequently denies having sent the message P to Bob. When the case comes up in court, Bob can produce both P and $D_A(P)$. The judge can easily verify that Bob indeed has a valid message encrypted by D_A by simply applying E_A to it. Since Bob does not

know what Alice's private key is, the only way Bob could have acquired a message encrypted by it is if Alice did indeed send it. While in jail for perjury and fraud, Alice will have much time to devise interesting new public-key algorithms.

Although using public-key cryptography for digital signatures is an elegant scheme, there are problems that are related to the environment in which they operate rather than to the basic algorithm. For one thing, Bob can prove that a message was sent by Alice only as long as D_A remains secret. If Alice discloses her secret key, the argument no longer holds, because anyone could have sent the message, including Bob himself.

The problem might arise, for example, if Bob is Alice's stockbroker. Suppose that Alice tells Bob to buy a certain stock or bond. Immediately thereafter, the price drops sharply. To repudiate her message to Bob, Alice runs to the police claiming that her home was burglarized and the PC holding her key was stolen. Depending on the laws in her state or country, she may or may not be legally liable, especially if she claims not to have discovered the break-in until getting home from work, several hours after it allegedly happened.

Another problem with the signature scheme is what happens if Alice decides to change her key. Doing so is clearly legal, and it is probably a good idea to do so periodically. If a court case later arises, as described above, the judge will apply the *current* E_A to $D_A(P)$ and discover that it does not produce P. Bob will look pretty stupid at this point.

In principle, any public-key algorithm can be used for digital signatures. The de facto industry standard is the RSA algorithm. Many security products use it. However, in 1991, NIST proposed using a variant of the El Gamal public-key algorithm for its new **Digital Signature Standard** (**DSS**). El Gamal gets its security from the difficulty of computing discrete logarithms, rather than from the difficulty of factoring large numbers.

As usual when the government tries to dictate cryptographic standards, there was an uproar. DSS was criticized for being

1. Too secret (NSA designed the protocol for using El Gamal).
2. Too slow (10 to 40 times slower than RSA for checking signatures).
3. Too new (El Gamal had not yet been thoroughly analyzed).
4. Too insecure (fixed 512-bit key).

In a subsequent revision, the fourth point was rendered moot when keys up to 1024 bits were allowed. Nevertheless, the first two points remain valid.

8.4.3 Message Digests

One criticism of signature methods is that they often couple two distinct functions: authentication and secrecy. Often, authentication is needed but secrecy is not always needed. Also, getting an export license is often easier if the system in

question provides only authentication but not secrecy. Below we will describe an authentication scheme that does not require encrypting the entire message.

This scheme is based on the idea of a one-way hash function that takes an arbitrarily long piece of plaintext and from it computes a fixed-length bit string. This hash function, *MD*, often called a **message digest**, has four important properties:

1. Given P, it is easy to compute $MD(P)$.
2. Given $MD(P)$, it is effectively impossible to find P.
3. Given P, no one can find P' such that $MD(P') = MD(P)$.
4. A change to the input of even 1 bit produces a very different output.

To meet criterion 3, the hash should be at least 128 bits long, preferably more. To meet criterion 4, the hash must mangle the bits very thoroughly, not unlike the symmetric-key encryption algorithms we have seen.

Computing a message digest from a piece of plaintext is much faster than encrypting that plaintext with a public-key algorithm, so message digests can be used to speed up digital signature algorithms. To see how this works, consider the signature protocol of Fig. 8-18 again. Instead, of signing P with $K_{BB}(A, t, P)$, *BB* now computes the message digest by applying *MD* to P, yielding $MD(P)$. *BB* then encloses $K_{BB}(A, t, MD(P))$ as the fifth item in the list encrypted with K_B that is sent to Bob, instead of $K_{BB}(A, t, P)$.

If a dispute arises, Bob can produce both P and $K_{BB}(A, t, MD(P))$. After Big Brother has decrypted it for the judge, Bob has $MD(P)$, which is guaranteed to be genuine, and the alleged P. However, since it is effectively impossible for Bob to find any other message that gives this hash, the judge will easily be convinced that Bob is telling the truth. Using message digests in this way saves both encryption time and message transport costs.

Message digests work in public-key cryptosystems, too, as shown in Fig. 8-20. Here, Alice first computes the message digest of her plaintext. She then signs the message digest and sends both the signed digest and the plaintext to Bob. If Trudy replaces P along the way, Bob will see this when he computes $MD(P)$.

Figure 8-20. Digital signatures using message digests.

SHA-1 and SHA-2

A variety of message digest functions have been proposed. One of the most widely used functions is **SHA-1** (**Secure Hash Algorithm 1**) (NIST, 1993). Like all message digests, it operates by mangling bits in a sufficiently complicated way that every output bit is affected by every input bit. SHA-1 was developed by NSA and blessed by NIST in FIPS 180-1. It processes input data in 512-bit blocks, and it generates a 160-bit message digest. A typical way for Alice to send a nonsecret but signed message to Bob is illustrated in Fig. 8-21. Here, her plaintext message is fed into the SHA-1 algorithm to get a 160-bit SHA-1 hash. Alice then signs the hash with her RSA private key and sends both the plaintext message and the signed hash to Bob.

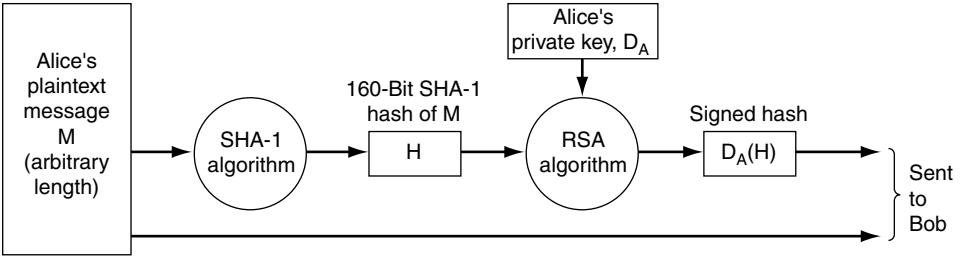


Figure 8-21. Use of SHA-1 and RSA for signing nonsecret messages.

After receiving the message, Bob computes the SHA-1 hash himself and also applies Alice's public key to the signed hash to get the original hash, *H*. If the two agree, the message is considered valid. Since there is no way for Trudy to modify the (plaintext) message while it is in transit and produce a new one that hashes to *H*, Bob can easily detect any changes Trudy has made to the message. For messages whose integrity is important but whose contents are not secret, the scheme of Fig. 8-21 is widely used. For a relatively small cost in computation, it guarantees that any modifications made to the plaintext message in transit can be detected with very high probability.

Now let us briefly see how SHA-1 works. It starts out by padding the message by adding a 1 bit to the end, followed by as many 0 bits as are necessary, but at least 64, to make the length a multiple of 512 bits. Then a 64-bit number containing the message length before padding is ORed into the low-order 64 bits. In Fig. 8-22, the message is shown with padding on the right because English text and figures go from left to right (i.e., the lower right is generally perceived as the end of the figure). With computers, this orientation corresponds to big-endian machines such as the SPARC and the IBM 360 and its successors, but SHA-1 always pads the end of the message, no matter which endian machine is used.

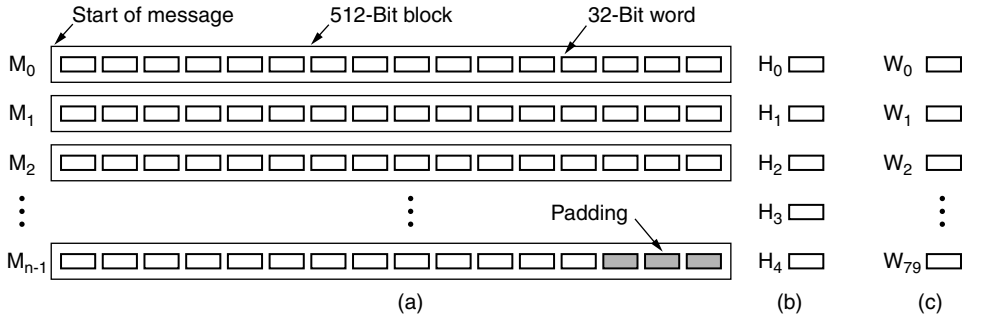


Figure 8-22. (a) A message padded out to a multiple of 512 bits. (b) The output variables. (c) The word array.

During the computation, SHA-1 maintains five 32-bit variables, H_0 through H_4, where the hash accumulates. These are shown in Fig. 8-22(b). They are initialized to constants specified in the standard.

Each of the blocks M_0 through M_{n-1} is now processed in turn. For the current block, the 16 words are first copied into the start of an auxiliary 80-word array, W, as shown in Fig. 8-22(c). Then the other 64 words in W are filled in using the formula

$$W_i = S^1(W_{i-3} \text{ XOR } W_{i-8} \text{ XOR } W_{i-14} \text{ XOR } W_{i-16}) \quad (16 \le i \le 79)$$

where $S^b(W)$ represents the left circular rotation of the 32-bit word, W, by b bits. Now five scratch variables, A through E, are initialized from H_0 through H_4, respectively.

The actual calculation can be expressed in pseudo-C as

```
for (i = 0; i < 80; i++) {
    temp = S^5(A) + f_i (B, C, D) + E + W_i + K_i;
    E = D;  D = C;  C = S^30(B);  B = A;  A = temp;
}
```

where the K_i constants are defined in the standard. The mixing functions f_i are defined as

$f_i(B,C,D) = (B \text{ AND } C) \text{ OR } (\text{NOT } B \text{ AND } D)$ $(0 \le i \le 19)$
$f_i(B,C,D) = B \text{ XOR } C \text{ XOR } D$ $(20 \le i \le 39)$
$f_i(B,C,D) = (B \text{ AND } C) \text{ OR } (B \text{ AND } D) \text{ OR } (C \text{ AND } D)$ $(40 \le i \le 59)$
$f_i(B,C,D) = B \text{ XOR } C \text{ XOR } D$ $(60 \le i \le 79)$

When all 80 iterations of the loop are completed, A through E are added to H_0 through H_4, respectively.

Now that the first 512-bit block has been processed, the next one is started. The W array is reinitialized from the new block, but H is left as it was. When this

block is finished, the next one is started, and so on, until all the 512-bit message blocks have been tossed into the soup. When the last block has been finished, the five 32-bit words in the *H* array are output as the 160-bit cryptographic hash. The complete C code for SHA-1 is given in RFC 3174.

New versions of SHA-1 have been developed that produce hashes of 224, 256, 384, and 512 bits. Collectively, these versions are called SHA-2. Not only are these hashes longer than SHA-1 hashes, but the digest function has been changed to combat some potential weaknesses of SHA-1. SHA-2 is not yet widely used, but it is likely to be in the future.

MD5

For completeness, we will mention another digest that is popular. **MD5** (Rivest, 1992) is the fifth in a series of message digests designed by Ronald Rivest. Very briefly, the message is padded to a length of 448 bits (modulo 512). Then the original length of the message is appended as a 64-bit integer to give a total input whose length is a multiple of 512 bits. Each round of the computation takes a 512-bit block of input and mixes it thoroughly with a running 128-bit buffer. For good measure, the mixing uses a table constructed from the sine function. The point of using a known function is to avoid any suspicion that the designer built in a clever back door through which only he can enter. This process continues until all the input blocks have been consumed. The contents of the 128-bit buffer form the message digest.

After more than a decade of solid use and study, weaknesses in MD5 have led to the ability to find collisions, or different messages with the same hash (Sotirov, et al., 2008). This is the death knell for a digest function because it means that the digest cannot safely be used to represent a message. Thus, the security community considers MD5 to be broken; it should be replaced where possible and no new systems should use it as part of their design. Nevertheless, you may still see MD5 used in existing systems.

8.4.4 The Birthday Attack

In the world of crypto, nothing is ever what it seems to be. One might think that it would take on the order of 2^m operations to subvert an m-bit message digest. In fact, $2^{m/2}$ operations will often do using the **birthday attack,** an approach published by Yuval (1979) in his now-classic paper "How to Swindle Rabin."

The idea for this attack comes from a technique that math professors often use in their probability courses. The question is: how many students do you need in a class before the probability of having two people with the same birthday exceeds 1/2? Most students expect the answer to be way over 100. In fact, probability theory says it is just 23. Without giving a rigorous analysis, intuitively, with 23

people, we can form $(23 \times 22)/2 = 253$ different pairs, each of which has a probability of 1/365 of being a hit. In this light, it is not really so surprising any more.

More generally, if there is some mapping between inputs and outputs with n inputs (people, messages, etc.) and k possible outputs (birthdays, message digests, etc.), there are $n(n-1)/2$ input pairs. If $n(n-1)/2 > k$, the chance of having at least one match is pretty good. Thus, approximately, a match is likely for $n > \sqrt{k}$. This result means that a 64-bit message digest can probably be broken by generating about 2^{32} messages and looking for two with the same message digest.

Let us look at a practical example. The Department of Computer Science at State University has one position for a tenured faculty member and two candidates, Tom and Dick. Tom was hired two years before Dick, so he goes up for review first. If he gets it, Dick is out of luck. Tom knows that the department chairperson, Marilyn, thinks highly of his work, so he asks her to write him a letter of recommendation to the Dean, who will decide on Tom's case. Once sent, all letters become confidential.

Marilyn tells her secretary, Ellen, to write the Dean a letter, outlining what she wants in it. When it is ready, Marilyn will review it, compute and sign the 64-bit digest, and send it to the Dean. Ellen can send the letter later by email.

Unfortunately for Tom, Ellen is romantically involved with Dick and would like to do Tom in, so she writes the following letter with the 32 bracketed options:

Dear Dean Smith,

This [*letter* | *message*] is to give my [*honest* | *frank*] opinion of Prof. Tom Wilson, who is [*a candidate* | *up*] for tenure [*now* | *this year*]. I have [*known* | *worked with*] Prof. Wilson for [*about* | *almost*] six years. He is an [*outstanding* | *excellent*] researcher of great [*talent* | *ability*] known [*worldwide* | *internationally*] for his [*brilliant* | *creative*] insights into [*many* | *a wide variety of*] [*difficult* | *challenging*] problems.

He is also a [*highly* | *greatly*] [*respected* | *admired*] [*teacher* | *educator*]. His students give his [*classes* | *courses*] [*rave* | *spectacular*] reviews. He is [*our* | *the Department's*] [*most popular* | *best-loved*] [*teacher* | *instructor*].

[*In addition* | *Additionally*] Prof. Wilson is a [*gifted* | *effective*] fund raiser. His [*grants* | *contracts*] have brought a [*large* | *substantial*] amount of money into [*the* | *our*] Department. [*This money has* | *These funds have*] [*enabled* | *permitted*] us to [*pursue* | *carry out*] many [*special* | *important*] programs, [*such as* | *for example*] your State 2000 program. Without these funds we would [*be unable* | *not be able*] to continue this program, which is so [*important* | *essential*] to both of us. I strongly urge you to grant him tenure.

Unfortunately for Tom, as soon as Ellen finishes composing and typing in this letter, she also writes a second one:

Dear Dean Smith,

This [*letter* | *message*] is to give my [*honest* | *frank*] opinion of Prof. Tom Wilson, who is [*a candidate* | *up*] for tenure [*now* | *this year*]. I have [*known* | *worked with*] Tom for [*about* | *almost*] six years. He is a [*poor* | *weak*] researcher not well known in his [*field* | *area*]. His research [*hardly ever* | *rarely*] shows [*insight in* | *understanding of*] the [*key* | *major*] problems of [*the* | *our*] day.

Furthermore, he is not a [*respected* | *admired*] [*teacher* | *educator*]. His students give his [*classes* | *courses*] [*poor* | *bad*] reviews. He is [*our* | *the Department's*] least popular [*teacher* | *instructor*], known [*mostly* | *primarily*] within [*the* | *our*] Department for his [*tendency* | *propensity*] to [*ridicule* | *embarrass*] students [*foolish* | *imprudent*] enough to ask questions in his classes.

[*In addition* | *Additionally*] Tom is a [*poor* | *marginal*] fund raiser. His [*grants* | *contracts*] have brought only a [*meager* | *insignificant*] amount of money into [*the* | *our*] Department. Unless new [*money is* | *funds are*] quickly located, we may have to cancel some essential programs, such as your State 2000 program. Unfortunately, under these [*conditions* | *circumstances*] I cannot in good [*conscience* | *faith*] recommend him to you for [*tenure* | *a permanent position*].

Now Ellen programs her computer to compute the 2^{32} message digests of each letter overnight. Chances are, one digest of the first letter will match one digest of the second. If not, she can add a few more options and try again tonight. Suppose that she finds a match. Call the "good" letter *A* and the "bad" one *B*.

Ellen now emails letter *A* to Marilyn for approval. Letter *B* she keeps secret, showing it to no one. Marilyn, of course, approves it, computes her 64-bit message digest, signs the digest, and emails the signed digest off to Dean Smith. Independently, Ellen emails letter *B* to the Dean (not letter *A*, as she is supposed to).

After getting the letter and signed message digest, the Dean runs the message digest algorithm on letter *B*, sees that it agrees with what Marilyn sent him, and fires Tom. The Dean does not realize that Ellen managed to generate two letters with the same message digest and sent her a different one than the one Marilyn saw and approved. (Optional ending: Ellen tells Dick what she did. Dick is appalled and breaks off the affair. Ellen is furious and confesses to Marilyn. Marilyn calls the Dean. Tom gets tenure after all.) With SHA-1, the birthday attack is difficult because even at the ridiculous speed of 1 trillion digests per second, it would take over 32,000 years to compute all 2^{80} digests of two letters with 80 variants each, and even then a match is not guaranteed. With a cloud of 1,000,000 chips working in parallel, 32,000 years becomes 2 weeks.

8.5 MANAGEMENT OF PUBLIC KEYS

Public-key cryptography makes it possible for people who do not share a common key in advance to nevertheless communicate securely. It also makes signing messages possible without the presence of a trusted third party. Finally,

signed message digests make it possible for the recipient to verify the integrity of received messages easily and securely.

However, there is one problem that we have glossed over a bit too quickly: if Alice and Bob do not know each other, how do they get each other's public keys to start the communication process? The obvious solution—put your public key on your Web site—does not work, for the following reason. Suppose that Alice wants to look up Bob's public key on his Web site. How does she do it? She starts by typing in Bob's URL. Her browser then looks up the DNS address of Bob's home page and sends it a *GET* request, as shown in Fig. 8-23. Unfortunately, Trudy intercepts the request and replies with a fake home page, probably a copy of Bob's home page except for the replacement of Bob's public key with Trudy's public key. When Alice now encrypts her first message with E_T, Trudy decrypts it, reads it, re-encrypts it with Bob's public key, and sends it to Bob, who is none the wiser that Trudy is reading his incoming messages. Worse yet, Trudy could modify the messages before reencrypting them for Bob. Clearly, some mechanism is needed to make sure that public keys can be exchanged securely.

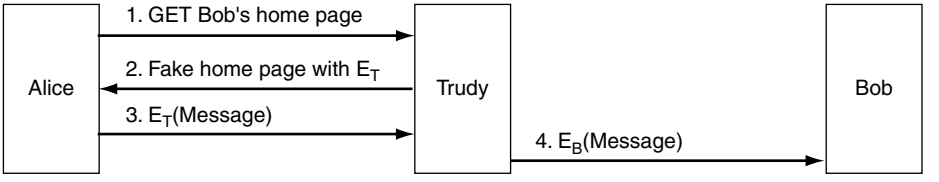


Figure 8-23. A way for Trudy to subvert public-key encryption.

8.5.1 Certificates

As a first attempt at distributing public keys securely, we could imagine a **KDC key distribution center** available online 24 hours a day to provide public keys on demand. One of the many problems with this solution is that it is not scalable, and the key distribution center would rapidly become a bottleneck. Also, if it ever went down, Internet security would suddenly grind to a halt.

For these reasons, people have developed a different solution, one that does not require the key distribution center to be online all the time. In fact, it does not have to be online at all. Instead, what it does is certify the public keys belonging to people, companies, and other organizations. An organization that certifies public keys is now called a **CA** (**Certification Authority**).

As an example, suppose that Bob wants to allow Alice and other people he does not know to communicate with him securely. He can go to the CA with his public key along with his passport or driver's license and ask to be certified. The CA then issues a certificate similar to the one in Fig. 8-24 and signs its SHA-1

hash with the CA's private key. Bob then pays the CA's fee and gets a CD-ROM containing the certificate and its signed hash.

Figure 8-24. A possible certificate and its signed hash.

The fundamental job of a certificate is to bind a public key to the name of a principal (individual, company, etc.). Certificates themselves are not secret or protected. Bob might, for example, decide to put his new certificate on his Web site, with a link on the main page saying: Click here for my public-key certificate. The resulting click would return both the certificate and the signature block (the signed SHA-1 hash of the certificate).

Now let us run through the scenario of Fig. 8-23 again. When Trudy intercepts Alice's request for Bob's home page, what can she do? She can put her own certificate and signature block on the fake page, but when Alice reads the contents of the certificate she will immediately see that she is not talking to Bob because Bob's name is not in it. Trudy can modify Bob's home page on the fly, replacing Bob's public key with her own. However, when Alice runs the SHA-1 algorithm on the certificate, she will get a hash that does not agree with the one she gets when she applies the CA's well-known public key to the signature block. Since Trudy does not have the CA's private key, she has no way of generating a signature block that contains the hash of the modified Web page with her public key on it. In this way, Alice can be sure she has Bob's public key and not Trudy's or someone else's. And as we promised, this scheme does not require the CA to be online for verification, thus eliminating a potential bottleneck.

While the standard function of a certificate is to bind a public key to a principal, a certificate can also be used to bind a public key to an **attribute**. For example, a certificate could say: "This public key belongs to someone over 18." It could be used to prove that the owner of the private key was not a minor and thus allowed to access material not suitable for children, and so on, but without disclosing the owner's identity. Typically, the person holding the certificate would send it to the Web site, principal, or process that cared about age. That site, principal, or process would then generate a random number and encrypt it with the public key in the certificate. If the owner were able to decrypt it and send it back,

that would be proof that the owner indeed had the attribute stated in the certificate. Alternatively, the random number could be used to generate a session key for the ensuing conversation.

Another example of where a certificate might contain an attribute is in an object-oriented distributed system. Each object normally has multiple methods. The owner of the object could provide each customer with a certificate giving a bit map of which methods the customer is allowed to invoke and binding the bit map to a public key using a signed certificate. Again, if the certificate holder can prove possession of the corresponding private key, he will be allowed to perform the methods in the bit map. This approach has the property that the owner's identity need not be known, a property useful in situations where privacy is important.

8.5.2 X.509

If everybody who wanted something signed went to the CA with a different kind of certificate, managing all the different formats would soon become a problem. To solve this problem, a standard for certificates has been devised and approved by ITU. The standard is called **X.509** and is in widespread use on the Internet. It has gone through three versions since the initial standardization in 1988. We will discuss V3.

X.509 has been heavily influenced by the OSI world, borrowing some of its worst features (e.g., naming and encoding). Surprisingly, IETF went along with X.509, even though in nearly every other area, from machine addresses to transport protocols to email formats, IETF generally ignored OSI and tried to do it right. The IETF version of X.509 is described in RFC 5280.

At its core, X.509 is a way to describe certificates. The primary fields in a certificate are listed in Fig. 8-25. The descriptions given there should provide a general idea of what the fields do. For additional information, please consult the standard itself or RFC 2459.

For example, if Bob works in the loan department of the Money Bank, his X.500 address might be

```
/C=US/O=MoneyBank/OU=Loan/CN=Bob/
```

where *C* is for country, *O* is for organization, *OU* is for organizational unit, and *CN* is for common name. CAs and other entities are named in a similar way. A substantial problem with X.500 names is that if Alice is trying to contact *bob@moneybank.com* and is given a certificate with an X.500 name, it may not be obvious to her that the certificate refers to the Bob she wants. Fortunately, starting with version 3, DNS names are now permitted instead of X.500 names, so this problem may eventually vanish.

Certificates are encoded using OSI **ASN.1** (**Abstract Syntax Notation 1**), which is sort of like a struct in C, except with a extremely peculiar and verbose notation. More information about X.509 is given by Ford and Baum (2000).

Field	Meaning
Version	Which version of X.509
Serial number	This number plus the CA's name uniquely identifies the certificate
Signature algorithm	The algorithm used to sign the certificate
Issuer	X.500 name of the CA
Validity period	The starting and ending times of the validity period
Subject name	The entity whose key is being certified
Public key	The subject's public key and the ID of the algorithm using it
Issuer ID	An optional ID uniquely identifying the certificate's issuer
Subject ID	An optional ID uniquely identifying the certificate's subject
Extensions	Many extensions have been defined
Signature	The certificate's signature (signed by the CA's private key)

Figure 8-25. The basic fields of an X.509 certificate.

8.5.3 Public Key Infrastructures

Having a single CA to issue all the world's certificates obviously would not work. It would collapse under the load and be a central point of failure as well. A possible solution might be to have multiple CAs, all run by the same organization and all using the same private key to sign certificates. While this would solve the load and failure problems, it introduces a new problem: key leakage. If there were dozens of servers spread around the world, all holding the CA's private key, the chance of the private key being stolen or otherwise leaking out would be greatly increased. Since the compromise of this key would ruin the world's electronic security infrastructure, having a single central CA is very risky.

In addition, which organization would operate the CA? It is hard to imagine any authority that would be accepted worldwide as legitimate and trustworthy. In some countries, people would insist that it be a government, while in other countries they would insist that it not be a government.

For these reasons, a different way for certifying public keys has evolved. It goes under the general name of **PKI** (**Public Key Infrastructure**). In this section, we will summarize how it works in general, although there have been many proposals, so the details will probably evolve in time.

A PKI has multiple components, including users, CAs, certificates, and directories. What the PKI does is provide a way of structuring these components and define standards for the various documents and protocols. A particularly simple form of PKI is a hierarchy of CAs, as depicted in Fig. 8-26. In this example we have shown three levels, but in practice there might be fewer or more. The top-level CA, the root, certifies second-level CAs, which we here call **RA**s (**Regional**

Authorities) because they might cover some geographic region, such as a country or continent. This term is not standard, though; in fact, no term is really standard for the different levels of the tree. These in turn certify the real CAs, which issue the X.509 certificates to organizations and individuals. When the root authorizes a new RA, it generates an X.509 certificate stating that it has approved the RA, includes the new RA's public key in it, signs it, and hands it to the RA. Similarly, when an RA approves a new CA, it produces and signs a certificate stating its approval and containing the CA's public key.

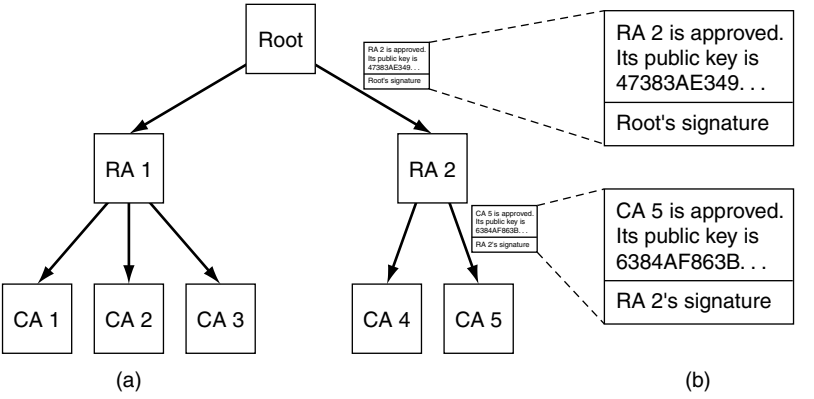


Figure 8-26. (a) A hierarchical PKI. (b) A chain of certificates.

Our PKI works like this. Suppose that Alice needs Bob's public key in order to communicate with him, so she looks for and finds a certificate containing it, signed by CA 5. But Alice has never heard of CA 5. For all she knows, CA 5 might be Bob's 10-year-old daughter. She could go to CA 5 and say: "Prove your legitimacy." CA 5 will respond with the certificate it got from RA 2, which contains CA 5's public key. Now armed with CA 5's public key, she can verify that Bob's certificate was indeed signed by CA 5 and is thus legal.

Unless RA 2 is Bob's 12-year-old son. So, the next step is for her to ask RA 2 to prove it is legitimate. The response to her query is a certificate signed by the root and containing RA 2's public key. Now Alice is sure she has Bob's public key.

But how does Alice find the root's public key? Magic. It is assumed that everyone knows the root's public key. For example, her browser might have been shipped with the root's public key built in.

Bob is a friendly sort of guy and does not want to cause Alice a lot of work. He knows that she is going to have to check out CA 5 and RA 2, so to save her some trouble, he collects the two needed certificates and gives her the two certificates along with his. Now she can use her own knowledge of the root's public key to verify the top-level certificate and the public key contained therein to verify the second one. Alice does not need to contact anyone to do the verification.

Because the certificates are all signed, she can easily detect any attempts to tamper with their contents. A chain of certificates going back to the root like this is sometimes called a **chain of trust** or a **certification path**. The technique is widely used in practice.

Of course, we still have the problem of who is going to run the root. The solution is not to have a single root, but to have many roots, each with its own RAs and CAs. In fact, modern browsers come preloaded with the public keys for over 100 roots, sometimes referred to as **trust anchors**. In this way, having a single worldwide trusted authority can be avoided.

But there is now the issue of how the browser vendor decides which purported trust anchors are reliable and which are sleazy. It all comes down to the user trusting the browser vendor to make wise choices and not simply approve all trust anchors willing to pay its inclusion fee. Most browsers allow users to inspect the root keys (usually in the form of certificates signed by the root) and delete any that seem shady.

Directories

Another issue for any PKI is where certificates (and their chains back to some known trust anchor) are stored. One possibility is to have each user store his or her own certificates. While doing this is safe (i.e., there is no way for users to tamper with signed certificates without detection), it is also inconvenient. One alternative that has been proposed is to use DNS as a certificate directory. Before contacting Bob, Alice probably has to look up his IP address using DNS, so why not have DNS return Bob's entire certificate chain along with his IP address?

Some people think this is the way to go, but others would prefer dedicated directory servers whose only job is managing X.509 certificates. Such directories could provide lookup services by using properties of the X.500 names. For example, in theory such a directory service could answer a query such as: "Give me a list of all people named Alice who work in sales departments anywhere in the U.S. or Canada.''

Revocation

The real world is full of certificates, too, such as passports and drivers' licenses. Sometimes these certificates can be revoked, for example, drivers' licenses can be revoked for drunken driving and other driving offenses. The same problem occurs in the digital world: the grantor of a certificate may decide to revoke it because the person or organization holding it has abused it in some way. It can also be revoked if the subject's private key has been exposed or, worse yet, the CA's private key has been compromised. Thus, a PKI needs to deal with the issue of revocation. The possibility of revocation complicates matters.

A first step in this direction is to have each CA periodically issue a **CRL** (**Certificate Revocation List**) giving the serial numbers of all certificates that it has revoked. Since certificates contain expiry times, the CRL need only contain the serial numbers of certificates that have not yet expired. Once its expiry time has passed, a certificate is automatically invalid, so no distinction is needed between those that just timed out and those that were actually revoked. In both cases, they cannot be used any more.

Unfortunately, introducing CRLs means that a user who is about to use a certificate must now acquire the CRL to see if the certificate has been revoked. If it has been, it should not be used. However, even if the certificate is not on the list, it might have been revoked just after the list was published. Thus, the only way to really be sure is to ask the CA. And on the next use of the same certificate, the CA has to be asked again, since the certificate might have been revoked a few seconds ago.

Another complication is that a revoked certificate could conceivably be reinstated, for example, if it was revoked for nonpayment of some fee that has since been paid. Having to deal with revocation (and possibly reinstatement) eliminates one of the best properties of certificates, namely, that they can be used without having to contact a CA.

Where should CRLs be stored? A good place would be the same place the certificates themselves are stored. One strategy is for the CA to actively push out CRLs periodically and have the directories process them by simply removing the revoked certificates. If directories are not used for storing certificates, the CRLs can be cached at various places around the network. Since a CRL is itself a signed document, if it is tampered with, that tampering can be easily detected.

If certificates have long lifetimes, the CRLs will be long, too. For example, if credit cards are valid for 5 years, the number of revocations outstanding will be much longer than if new cards are issued every 3 months. A standard way to deal with long CRLs is to issue a master list infrequently, but issue updates to it more often. Doing this reduces the bandwidth needed for distributing the CRLs.

8.6 COMMUNICATION SECURITY

We have now finished our study of the tools of the trade. Most of the important techniques and protocols have been covered. The rest of the chapter is about how these techniques are applied in practice to provide network security, plus some thoughts about the social aspects of security at the end of the chapter.

In the following four sections, we will look at communication security, that is, how to get the bits secretly and without modification from source to destination and how to keep unwanted bits outside the door. These are by no means the only security issues in networking, but they are certainly among the most important ones, making this a good place to start our study.

8.6.1 IPsec

IETF has known for years that security was lacking in the Internet. Adding it was not easy because a war broke out about where to put it. Most security experts believe that to be really secure, encryption and integrity checks have to be end to end (i.e., in the application layer). That is, the source process encrypts and/or integrity protects the data and sends them to the destination process where they are decrypted and/or verified. Any tampering done in between these two processes, including within either operating system, can then be detected. The trouble with this approach is that it requires changing all the applications to make them security aware. In this view, the next best approach is putting encryption in the transport layer or in a new layer between the application layer and the transport layer, making it still end to end but not requiring applications to be changed.

The opposite view is that users do not understand security and will not be capable of using it correctly and nobody wants to modify existing programs in any way, so the network layer should authenticate and/or encrypt packets without the users being involved. After years of pitched battles, this view won enough support that a network layer security standard was defined. In part, the argument was that having network layer encryption does not prevent security-aware users from doing it right and it does help security-unaware users to some extent.

The result of this war was a design called **IPsec** (**IP security**), which is described in RFCs 2401, 2402, and 2406, among others. Not all users want encryption (because it is computationally expensive). Rather than make it optional, it was decided to require encryption all the time but permit the use of a null algorithm. The null algorithm is described and praised for its simplicity, ease of implementation, and great speed in RFC 2410.

The complete IPsec design is a framework for multiple services, algorithms, and granularities. The reason for multiple services is that not everyone wants to pay the price for having all the services all the time, so the services are available a la carte. The major services are secrecy, data integrity, and protection from replay attacks (where the intruder replays a conversation). All of these are based on symmetric-key cryptography because high performance is crucial.

The reason for having multiple algorithms is that an algorithm that is now thought to be secure may be broken in the future. By making IPsec algorithm-independent, the framework can survive even if some particular algorithm is later broken.

The reason for having multiple granularities is to make it possible to protect a single TCP connection, all traffic between a pair of hosts, or all traffic between a pair of secure routers, among other possibilities.

One slightly surprising aspect of IPsec is that even though it is in the IP layer, it is connection oriented. Actually, that is not so surprising because to have any security, a key must be established and used for some period of time—in essence, a kind of connection by a different name. Also, connections amortize the setup

costs over many packets. A "connection" in the context of IPsec is called an **SA** (**Security Association**). An SA is a simplex connection between two endpoints and has a security identifier associated with it. If secure traffic is needed in both directions, two security associations are required. Security identifiers are carried in packets traveling on these secure connections and are used to look up keys and other relevant information when a secure packet arrives.

Technically, IPsec has two principal parts. The first part describes two new headers that can be added to packets to carry the security identifier, integrity control data, and other information. The other part, **ISAKMP** (**Internet Security Association and Key Management Protocol**), deals with establishing keys. ISAKMP is a framework. The main protocol for carrying out the work is **IKE** (**Internet Key Exchange**). Version 2 of IKE as described in RFC 4306 should be used, as the earlier version was deeply flawed, as pointed out by Perlman and Kaufman (2000).

IPsec can be used in either of two modes. In **transport mode**, the IPsec header is inserted just after the IP header. The *Protocol* field in the IP header is changed to indicate that an IPsec header follows the normal IP header (before the TCP header). The IPsec header contains security information, primarily the SA identifier, a new sequence number, and possibly an integrity check of the payload.

In **tunnel mode**, the entire IP packet, header and all, is encapsulated in the body of a new IP packet with a completely new IP header. Tunnel mode is useful when the tunnel ends at a location other than the final destination. In some cases, the end of the tunnel is a security gateway machine, for example, a company firewall. This is commonly the case for a VPN (Virtual Private Network). In this mode, the security gateway encapsulates and decapsulates packets as they pass through it. By terminating the tunnel at this secure machine, the machines on the company LAN do not have to be aware of IPsec. Only the security gateway has to know about it.

Tunnel mode is also useful when a bundle of TCP connections is aggregated and handled as one encrypted stream because it prevents an intruder from seeing who is sending how many packets to whom. Sometimes just knowing how much traffic is going where is valuable information. For example, if during a military crisis, the amount of traffic flowing between the Pentagon and the White House were to drop sharply, but the amount of traffic between the Pentagon and some military installation deep in the Colorado Rocky Mountains were to increase by the same amount, an intruder might be able to deduce some useful information from these data. Studying the flow patterns of packets, even if they are encrypted, is called **traffic analysis**. Tunnel mode provides a way to foil it to some extent. The disadvantage of tunnel mode is that it adds an extra IP header, thus increasing packet size substantially. In contrast, transport mode does not affect packet size as much.

The first new header is **AH** (**Authentication Header**). It provides integrity checking and antireplay security, but not secrecy (i.e., no data encryption). The

use of AH in transport mode is illustrated in Fig. 8-27. In IPv4, it is interposed between the IP header (including any options) and the TCP header. In IPv6, it is just another extension header and is treated as such. In fact, the format is close to that of a standard IPv6 extension header. The payload may have to be padded out to some particular length for the authentication algorithm, as shown.

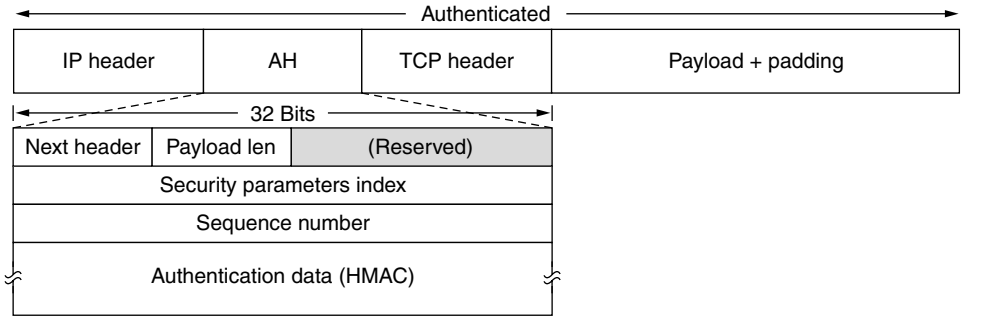


Figure 8-27. The IPsec authentication header in transport mode for IPv4.

Let us now examine the AH header. The *Next header* field is used to store the value that the IP *Protocol* field had before it was replaced with 51 to indicate that an AH header follows. In most cases, the code for TCP (6) will go here. The *Payload length* is the number of 32-bit words in the AH header minus 2.

The *Security parameters index* is the connection identifier. It is inserted by the sender to indicate a particular record in the receiver's database. This record contains the shared key used on this connection and other information about the connection. If this protocol had been invented by ITU rather than IETF, this field would have been called *Virtual circuit number*.

The *Sequence number* field is used to number all the packets sent on an SA. Every packet gets a unique number, even retransmissions. In other words, the retransmission of a packet gets a different number here than the original (even though its TCP sequence number is the same). The purpose of this field is to detect replay attacks. These sequence numbers may not wrap around. If all 2^{32} are exhausted, a new SA must be established to continue communication.

Finally, we come to *Authentication data*, which is a variable-length field that contains the payload's digital signature. When the SA is established, the two sides negotiate which signature algorithm they are going to use. Normally, public-key cryptography is not used here because packets must be processed extremely rapidly and all known public-key algorithms are too slow. Since IPsec is based on symmetric-key cryptography and the sender and receiver negotiate a shared key before setting up an SA, the shared key is used in the signature computation. One simple way is to compute the hash over the packet plus the shared key. The shared key is not transmitted, of course. A scheme like this is called an **HMAC**

(**Hashed Message Authentication Code**). It is much faster to compute than first running SHA-1 and then running RSA on the result.

The AH header does not allow encryption of the data, so it is mostly useful when integrity checking is needed but secrecy is not needed. One noteworthy feature of AH is that the integrity check covers some of the fields in the IP header, namely, those that do not change as the packet moves from router to router. The *Time to live* field changes on each hop, for example, so it cannot be included in the integrity check. However, the IP source address is included in the check, making it impossible for an intruder to falsify the origin of a packet.

The alternative IPsec header is **ESP** (**Encapsulating Security Payload**). Its use for both transport mode and tunnel mode is shown in Fig. 8-28.

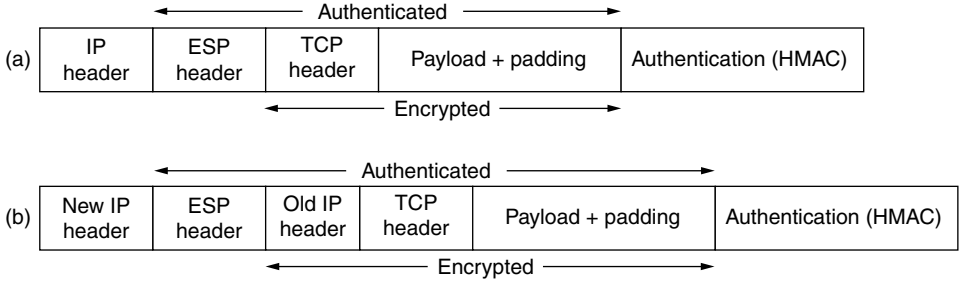


Figure 8-28. (a) ESP in transport mode. (b) ESP in tunnel mode.

The ESP header consists of two 32-bit words. They are the *Security parameters index* and *Sequence number* fields that we saw in AH. A third word that generally follows them (but is technically not part of the header) is the *Initialization vector* used for the data encryption, unless null encryption is used, in which case it is omitted.

ESP also provides for HMAC integrity checks, as does AH, but rather than being included in the header, they come after the payload, as shown in Fig. 8-28. Putting the HMAC at the end has an advantage in a hardware implementation: the HMAC can be calculated as the bits are going out over the network interface and appended to the end. This is why Ethernet and other LANs have their CRCs in a trailer, rather than in a header. With AH, the packet has to be buffered and the signature computed before the packet can be sent, potentially reducing the number of packets/sec that can be sent.

Given that ESP can do everything AH can do and more and is more efficient to boot, the question arises: why bother having AH at all? The answer is mostly historical. Originally, AH handled only integrity and ESP handled only secrecy. Later, integrity was added to ESP, but the people who designed AH did not want to let it die after all that work. Their only real argument is that AH checks part of the IP header, which ESP does not, but other than that it is really a weak argument. Another weak argument is that a product supporting AH but not ESP might

have less trouble getting an export license because it cannot do encryption. AH is likely to be phased out in the future.

8.6.2 Firewalls

The ability to connect any computer, anywhere, to any other computer, anywhere, is a mixed blessing. For individuals at home, wandering around the Internet is lots of fun. For corporate security managers, it is a nightmare. Most companies have large amounts of confidential information online—trade secrets, product development plans, marketing strategies, financial analyses, etc. Disclosure of this information to a competitor could have dire consequences.

In addition to the danger of information leaking out, there is also a danger of information leaking in. In particular, viruses, worms, and other digital pests can breach security, destroy valuable data, and waste large amounts of administrators' time trying to clean up the mess they leave. Often they are imported by careless employees who want to play some nifty new game.

Consequently, mechanisms are needed to keep "good" bits in and "bad" bits out. One method is to use IPsec. This approach protects data in transit between secure sites. However, IPsec does nothing to keep digital pests and intruders from getting onto the company LAN. To see how to accomplish this goal, we need to look at firewalls.

Firewalls are just a modern adaptation of that old medieval security standby: digging a deep moat around your castle. This design forced everyone entering or leaving the castle to pass over a single drawbridge, where they could be inspected by the I/O police. With networks, the same trick is possible: a company can have many LANs connected in arbitrary ways, but all traffic to or from the company is forced through an electronic drawbridge (firewall), as shown in Fig. 8-29. No other route exists.

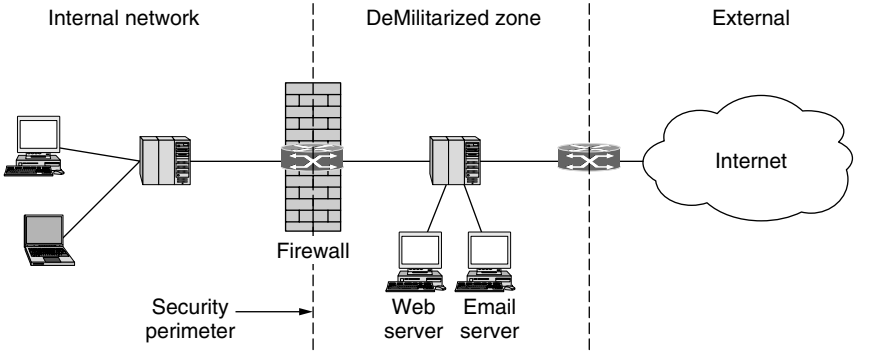


Figure 8-29. A firewall protecting an internal network.

The firewall acts as a **packet filter**. It inspects each and every incoming and outgoing packet. Packets meeting some criterion described in rules formulated by the network administrator are forwarded normally. Those that fail the test are uncermoniously dropped.

The filtering criterion is typically given as rules or tables that list sources and destinations that are acceptable, sources and destinations that are blocked, and default rules about what to do with packets coming from or going to other machines. In the common case of a TCP/IP setting, a source or destination might consist of an IP address and a port. Ports indicate which service is desired. For example, TCP port 25 is for mail, and TCP port 80 is for HTTP. Some ports can simply be blocked. For example, a company could block incoming packets for all IP addresses combined with TCP port 79. It was once popular for the Finger service to look up people's email addresses but is little used today.

Other ports are not so easily blocked. The difficulty is that network administrators want security but cannot cut off communication with the outside world. That arrangement would be much simpler and better for security, but there would be no end to user complaints about it. This is where arrangements such as the **DMZ** (**DeMilitarized Zone**) shown in Fig. 8-29 come in handy. The DMZ is the part of the company network that lies outside of the security perimeter. Anything goes here. By placing a machine such as a Web server in the DMZ, computers on the Internet can contact it to browse the company Web site. Now the firewall can be configured to block incoming TCP traffic to port 80 so that computers on the Internet cannot use this port to attack computers on the internal network. To allow the Web server to be managed, the firewall can have a rule to permit connections between internal machines and the Web server.

Firewalls have become much more sophisticated over time in an arms race with attackers. Originally, firewalls applied a rule set independently for each packet, but it proved difficult to write rules that allowed useful functionality but blocked all unwanted traffic. **Stateful firewalls** map packets to connections and use TCP/IP header fields to keep track of connections. This allows for rules that, for example, allow an external Web server to send packets to an internal host, but only if the internal host first establishes a connection with the external Web server. Such a rule is not possible with stateless designs that must either pass or drop all packets from the external Web server.

Another level of sophistication up from stateful processing is for the firewall to implement **application-level gateways**. This processing involves the firewall looking inside packets, beyond even the TCP header, to see what the application is doing. With this capability, it is possible to distinguish HTTP traffic used for Web browsing from HTTP traffic used for peer-to-peer file sharing. Administrators can write rules to spare the company from peer-to-peer file sharing but allow Web browsing that is vital for business. For all of these methods, outgoing traffic can be inspected as well as incoming traffic, for example, to prevent sensitive documents from being emailed outside of the company.

As the above discussion should make clear, firewalls violate the standard layering of protocols. They are network layer devices, but they peek at the transport and applications layers to do their filtering. This makes them fragile. For instance, firewalls tend to rely on standard port numbering conventions to determine what kind of traffic is carried in a packet. Standard ports are often used, but not by all computers, and not by all applications either. Some peer-to-peer applications select ports dynamically to avoid being easily spotted (and blocked). Encryption with IPSEC or other schemes hides higher-layer information from the firewall. Finally, a firewall cannot readily talk to the computers that communicate through it to tell them what policies are being applied and why their connection is being dropped. It must simply pretend to be a broken wire. For all these reasons, networking purists consider firewalls to be a blemish on the architecture of the Internet. However, the Internet can be a dangerous place if you are a computer. Firewalls help with that problem, so they are likely to stay.

Even if the firewall is perfectly configured, plenty of security problems still exist. For example, if a firewall is configured to allow in packets from only specific networks (e.g., the company's other plants), an intruder outside the firewall can put in false source addresses to bypass this check. If an insider wants to ship out secret documents, he can encrypt them or even photograph them and ship the photos as JPEG files, which bypasses any email filters. And we have not even discussed the fact that, although three-quarters of all attacks come from outside the firewall, the attacks that come from inside the firewall, for example, from disgruntled employees, are typically the most damaging (Verizon, 2009).

A different problem with firewalls is that they provide a single perimeter of defense. If that defense is breached, all bets are off. For this reason, firewalls are often used in a layered defense. For example, a firewall may guard the entrance to the internal network and each computer may also run its own firewall. Readers who think that one security checkpoint is enough clearly have not made an international flight on a scheduled airline recently.

In addition, there is a whole other class of attacks that firewalls cannot deal with. The basic idea of a firewall is to prevent intruders from getting in and secret data from getting out. Unfortunately, there are people who have nothing better to do than try to bring certain sites down. They do this by sending legitimate packets at the target in great numbers until it collapses under the load. For example, to cripple a Web site, an intruder can send a TCP *SYN* packet to establish a connection. The site will then allocate a table slot for the connection and send a *SYN + ACK* packet in reply. If the intruder does not respond, the table slot will be tied up for a few seconds until it times out. If the intruder sends thousands of connection requests, all the table slots will fill up and no legitimate connections will be able to get through. Attacks in which the intruder's goal is to shut down the target rather than steal data are called **DoS** (**Denial of Service**) attacks. Usually, the request packets have false source addresses so the intruder cannot be traced easily. DoS attacks against major Web sites are common on the Internet.

An even worse variant is one in which the intruder has already broken into hundreds of computers elsewhere in the world, and then commands all of them to attack the same target at the same time. Not only does this approach increase the intruder's firepower, but it also reduces his chances of detection since the packets are coming from a large number of machines belonging to unsuspecting users. Such an attack is called a **DDoS** (**Distributed Denial of Service**) attack. This attack is difficult to defend against. Even if the attacked machine can quickly recognize a bogus request, it does take some time to process and discard the request, and if enough requests per second arrive, the CPU will spend all its time dealing with them.

8.6.3 Virtual Private Networks

Many companies have offices and plants scattered over many cities, sometimes over multiple countries. In the olden days, before public data networks, it was common for such companies to lease lines from the telephone company between some or all pairs of locations. Some companies still do this. A network built up from company computers and leased telephone lines is called a **private network**.

Private networks work fine and are very secure. If the only lines available are the leased lines, no traffic can leak out of company locations and intruders have to physically wiretap the lines to break in, which is not easy to do. The problem with private networks is that leasing a dedicated T1 line between two points costs thousands of dollars a month, and T3 lines are many times more expensive. When public data networks and later the Internet appeared, many companies wanted to move their data (and possibly voice) traffic to the public network, but without giving up the security of the private network.

This demand soon led to the invention of **VPN**s (**Virtual Private Networks**), which are overlay networks on top of public networks but with most of the properties of private networks. They are called "virtual" because they are merely an illusion, just as virtual circuits are not real circuits and virtual memory is not real memory.

One popular approach is to build VPNs directly over the Internet. A common design is to equip each office with a firewall and create tunnels through the Internet between all pairs of offices, as illustrated in Fig. 8-30(a). A further advantage of using the Internet for connectivity is that the tunnels can be set up on demand to include, for example, the computer of an employee who is at home or traveling as long as the person has an Internet connection. This flexibility is much greater then is provided with leased lines, yet from the perspective of the computers on the VPN, the topology looks just like the private network case, as shown in Fig. 8-30(b). When the system is brought up, each pair of firewalls has to negotiate the parameters of its SA, including the services, modes, algorithms, and keys. If IPsec is used for the tunneling, it is possible to aggregate all traffic between any

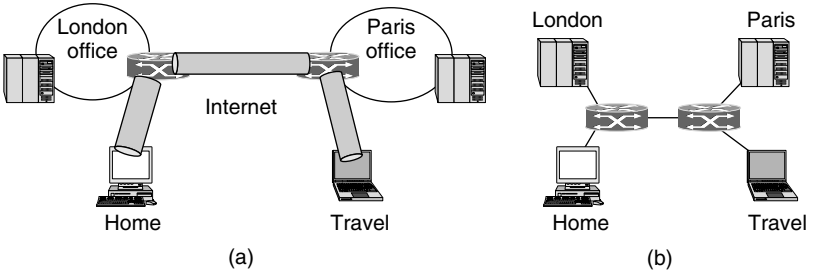


Figure 8-30. (a) A virtual private network. (b) Topology as seen from the inside.

two pairs of offices onto a single authenticated, encrypted SA, thus providing integrity control, secrecy, and even considerable immunity to traffic analysis. Many firewalls have VPN capabilities built in. Some ordinary routers can do this as well, but since firewalls are primarily in the security business, it is natural to have the tunnels begin and end at the firewalls, providing a clear separation between the company and the Internet. Thus, firewalls, VPNs, and IPsec with ESP in tunnel mode are a natural combination and widely used in practice.

Once the SAs have been established, traffic can begin flowing. To a router within the Internet, a packet traveling along a VPN tunnel is just an ordinary packet. The only thing unusual about it is the presence of the IPsec header after the IP header, but since these extra headers have no effect on the forwarding process, the routers do not care about this extra header.

Another approach that is gaining popularity is to have the ISP set up the VPN. Using MPLS (as discussed in Chap. 5), paths for the VPN traffic can be set up across the ISP network between the company offices. These paths keep the VPN traffic separate from other Internet traffic and can be guaranteed a certain amount of bandwidth or other quality of service.

A key advantage of a VPN is that it is completely transparent to all user software. The firewalls set up and manage the SAs. The only person who is even aware of this setup is the system administrator who has to configure and manage the security gateways, or the ISP administrator who has to configure the MPLS paths. To everyone else, it is like having a leased-line private network again. For more about VPNs, see Lewis (2006).

8.6.4 Wireless Security

It is surprisingly easy to design a system using VPNs and firewalls that is logically completely secure but that, in practice, leaks like a sieve. This situation can occur if some of the machines are wireless and use radio communication, which passes right over the firewall in both directions. The range of 802.11 networks is

often a few hundred meters, so anyone who wants to spy on a company can simply drive into the employee parking lot in the morning, leave an 802.11-enabled notebook computer in the car to record everything it hears, and take off for the day. By late afternoon, the hard disk will be full of valuable goodies. Theoretically, this leakage is not supposed to happen. Theoretically, people are not supposed to rob banks, either.

Much of the security problem can be traced to the manufacturers of wireless base stations (access points) trying to make their products user friendly. Usually, if the user takes the device out of the box and plugs it into the electrical power socket, it begins operating immediately—nearly always with no security at all, blurting secrets to everyone within radio range. If it is then plugged into an Ethernet, all the Ethernet traffic suddenly appears in the parking lot as well. Wireless is a snooper's dream come true: free data without having to do any work. It therefore goes without saying that security is even more important for wireless systems than for wired ones. In this section, we will look at some ways wireless networks handle security. Some additional information is given by Nichols and Lekkas (2002).

802.11 Security

Part of the 802.11 standard, originally called **802.11i**, prescribes a data link-level security protocol for preventing a wireless node from reading or interfering with messages sent between another pair of wireless nodes. It also goes by the trade name **WPA2** (**WiFi Protected Access 2**). Plain WPA is an interim scheme that implements a subset of 802.11i. It should be avoided in favor of WPA2.

We will describe 802.11i shortly, but will first note that it is a replacement for **WEP** (**Wired Equivalent Privacy**), the first generation of 802.11 security protocols. WEP was designed by a networking standards committee, which is a completely different process than, for example, the way NIST selected the design of AES. The results were devastating. What was wrong with it? Pretty much everything from a security perspective as it turns out. For example, WEP encrypted data for confidentiality by XORing it with the output of a stream cipher. Unfortunately, weak keying arrangements meant that the output was often reused. This led to trivial ways to defeat it. As another example, the integrity check was based on a 32-bit CRC. That is an efficient code for detecting transmission errors, but it is not a cryptographically strong mechanism for defeating attackers.

These and other design flaws made WEP very easy to compromise. The first practical demonstration that WEP was broken came when Adam Stubblefield was an intern at AT&T (Stubblefield et al., 2002). He was able to code up and test an attack outlined by Fluhrer et al. (2001) in one week, of which most of the time was spent convincing management to buy him a WiFi card to use in his experiments. Software to crack WEP passwords within a minute is now freely available and the use of WEP is very strongly discouraged. While it does prevent casual

access it does not provide any real form of security. The 802.11i group was put together in a hurry when it was clear that WEP was seriously broken. It produced a formal standard by June 2004.

Now we will describe 802.11i, which does provide real security if it is set up and used properly. There are two common scenarios in which WPA2 is used. The first is a corporate setting, in which a company has a separate authentication server that has a username and password database that can be used to determine if a wireless client is allowed to access the network. In this setting, clients use standard protocols to authenticate themselves to the network. The main standards are **802.1X**, with which the access point lets the client carry on a dialogue with the authentication server and observes the result, and **EAP** (**Extensible Authentication Protocol**) (RFC 3748), which tells how the client and the authentication server interact. Actually, EAP is a framework and other standards define the protocol messages. However, we will not delve into the many details of this exchange because they do not much matter for an overview.

The second scenario is in a home setting in which there is no authentication server. Instead, there is a single shared password that is used by clients to access the wireless network. This setup is less complex than having an authentication server, which is why it is used at home and in small businesses, but it is less secure as well. The main difference is that with an authentication server each client gets a key for encrypting traffic that is not known by the other clients. With a single shared password, different keys are derived for each client, but all clients have the same password and can derive each others' keys if they want to.

The keys that are used to encrypt traffic are computed as part of an authentication handshake. The handshake happens right after the client associates with a wireless network and authenticates with an authentication server, if there is one. At the start of the handshake, the client has either the shared network password or its password for the authentication server. This password is used to derive a master key. However, the master key is not used directly to encrypt packets. It is standard cryptographic practice to derive a session key for each period of usage, to change the key for different sessions, and to expose the master key to observation as little as possible. It is this session key that is computed in the handshake.

The session key is computed with the four-packet handshake shown in Fig. 8-31. First, the AP (access point) sends a random number for identification. Random numbers used just once in security protocols like this one are called **nonces**, which is more-or-less a contraction of "number used once." The client also picks its own nonce. It uses the nonces, its MAC address and that of the AP, and the master key to compute a session key, K_S. The session key is split into portions, each of which is used for different purposes, but we have omitted this detail. Now the client has session keys, but the AP does not. So the client sends its nonce to the AP, and the AP performs the same computation to derive the same session keys. The nonces can be sent in the clear because the keys cannot be derived from them without extra, secret information. The message from the client is protected

with an integrity check called a **MIC** (**Message Integrity Check**) based on the session key. The AP can check that the MIC is correct, and so the message indeed must have come from the client, after it computes the session keys. A MIC is just another name for a message authentication code, as in an HMAC. The term MIC is often used instead for networking protocols because of the potential for confusion with MAC (Medium Access Control) addresses.

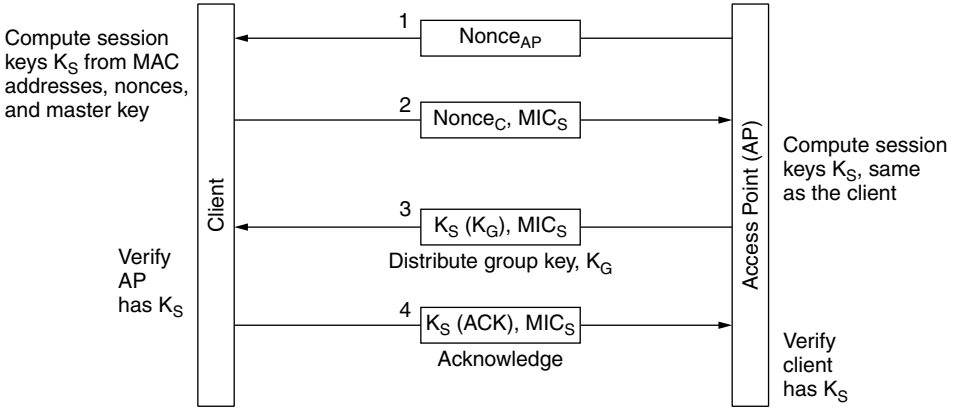


Figure 8-31. The 802.11i key setup handshake.

In the last two messages, the AP distributes a group key, K_G, to the client, and the client acknowledges the message. Receipt of these messages lets the client verify that the AP has the correct session keys, and vice versa. The group key is used for broadcast and multicast traffic on the 802.11 LAN. Because the result of the handshake is that every client has its own encryption keys, none of these keys can be used by the AP to broadcast packets to all of the wireless clients; a separate copy would need to be sent to each client using its key. Instead, a shared key is distributed so that broadcast traffic can be sent only once and received by all the clients. It must be updated as clients leave and join the network.

Finally, we get to the part where the keys are actually used to provide security. Two protocols can be used in 802.11i to provide message confidentiality, integrity, and authentication. Like WPA, one of the protocols, called **TKIP** (**Temporary Key Integrity Protocol**), was an interim solution. It was designed to improve security on old and slow 802.11 cards, so that at least some security that is better than WEP can be rolled out as a firmware upgrade. However, it, too, has now been broken so you are better off with the other, recommended protocol, **CCMP**. What does CCMP stand for? It is short for the somewhat spectacular name Counter mode with Cipher block chaining Message authentication code Protocol. We will just call it CCMP. You can call it anything you want.

CCMP works in a fairly straightforward way. It uses AES encryption with a 128-bit key and block size. The key comes from the session key. To provide confidentiality, messages are encrypted with AES in counter mode. Recall that we discussed cipher modes in Sec. 8.2.3. These modes are what prevent the same message from being encrypted to the same set of bits each time. Counter mode mixes a counter into the encryption. To provide integrity, the message, including header fields, is encrypted with cipher block chaining mode and the last 128-bit block is kept as the MIC. Then both the message (encrypted with counter mode) and the MIC are sent. The client and the AP can each perform this encryption, or verify this encryption when a wireless packet is received. For broadcast or multicast messages, the same procedure is used with the group key.

Bluetooth Security

Bluetooth has a considerably shorter range than 802.11, so it cannot easily be attacked from the parking lot, but security is still an issue here. For example, imagine that Alice's computer is equipped with a wireless Bluetooth keyboard. In the absence of security, if Trudy happened to be in the adjacent office, she could read everything Alice typed in, including all her outgoing email. She could also capture everything Alice's computer sent to the Bluetooth printer sitting next to it (e.g., incoming email and confidential reports). Fortunately, Bluetooth has an elaborate security scheme to try to foil the world's Trudies. We will now summarize the main features of it.

Bluetooth version 2.1 and later has four security modes, ranging from nothing at all to full data encryption and integrity control. As with 802.11, if security is disabled (the default for older devices), there is no security. Most users have security turned off until a serious breach has occurred; then they turn it on. In the agricultural world, this approach is known as locking the barn door after the horse has escaped.

Bluetooth provides security in multiple layers. In the physical layer, frequency hopping provides a tiny little bit of security, but since any Bluetooth device that moves into a piconet has to be told the frequency hopping sequence, this sequence is obviously not a secret. The real security starts when the newly arrived slave asks for a channel with the master. Before Bluetooth 2.1, two devices were assumed to share a secret key set up in advance. In some cases, both are hardwired by the manufacturer (e.g., for a headset and mobile phone sold as a unit). In other cases, one device (e.g., the headset) has a hardwired key and the user has to enter that key into the other device (e.g., the mobile phone) as a decimal number. These shared keys are called **passkeys**. Unfortunately, the passkeys are often hardcoded to "1234" or another predictable value, and in any case are four decimal digits, allowing only 10^4 choices. With simple secure pairing in Bluetooth 2.1, devices pick a code from a six-digit range, which makes the passkey much less predictable but still far from secure.

To establish a channel, the slave and master each check to see if the other one knows the passkey. If so, they negotiate whether that channel will be encrypted, integrity controlled, or both. Then they select a random 128-bit session key, some of whose bits may be public. The point of allowing this key weakening is to comply with government restrictions in various countries designed to prevent the export or use of keys longer than the government can break.

Encryption uses a stream cipher called $\mathbf{E_0}$; integrity control uses **SAFER+**. Both are traditional symmetric-key block ciphers. SAFER+ was submitted to the AES bake-off but was eliminated in the first round because it was slower than the other candidates. Bluetooth was finalized before the AES cipher was chosen; otherwise, it would most likely have used Rijndael.

The actual encryption using the stream cipher is shown in Fig. 8-14, with the plaintext XORed with the keystream to generate the ciphertext. Unfortunately, E_0 itself (like RC4) may have fatal weaknesses (Jakobsson and Wetzel, 2001). While it was not broken at the time of this writing, its similarities to the A5/1 cipher, whose spectacular failure compromises all GSM telephone traffic, are cause for concern (Biryukov et al., 2000). It sometimes amazes people (including the authors of this book), that in the perennial cat-and-mouse game between the cryptographers and the cryptanalysts, the cryptanalysts are so often on the winning side.

Another security issue is that Bluetooth authenticates only devices, not users, so theft of a Bluetooth device may give the thief access to the user's financial and other accounts. However, Bluetooth also implements security in the upper layers, so even in the event of a breach of link-level security, some security may remain, especially for applications that require a PIN code to be entered manually from some kind of keyboard to complete the transaction.

8.7 AUTHENTICATION PROTOCOLS

Authentication is the technique by which a process verifies that its communication partner is who it is supposed to be and not an imposter. Verifying the identity of a remote process in the face of a malicious, active intruder is surprisingly difficult and requires complex protocols based on cryptography. In this section, we will study some of the many authentication protocols that are used on insecure computer networks.

As an aside, some people confuse authorization with authentication. Authentication deals with the question of whether you are actually communicating with a specific process. Authorization is concerned with what that process is permitted to do. For example, say a client process contacts a file server and says: "I am Scott's process and I want to delete the file *cookbook.old*." From the file server's point of view, two questions must be answered:

1. Is this actually Scott's process (authentication)?
2. Is Scott allowed to delete *cookbook.old* (authorization)?

Only after both of these questions have been unambiguously answered in the affirmative can the requested action take place. The former question is really the key one. Once the file server knows to whom it is talking, checking authorization is just a matter of looking up entries in local tables or databases. For this reason, we will concentrate on authentication in this section.

The general model that essentially all authentication protocols use is this. Alice starts out by sending a message either to Bob or to a trusted **KDC** (**Key Distribution Center**), which is expected to be honest. Several other message exchanges follow in various directions. As these messages are being sent, Trudy may intercept, modify, or replay them in order to trick Alice and Bob or just to gum up the works.

Nevertheless, when the protocol has been completed, Alice is sure she is talking to Bob and Bob is sure he is talking to Alice. Furthermore, in most of the protocols, the two of them will also have established a secret **session key** for use in the upcoming conversation. In practice, for performance reasons, all data traffic is encrypted using symmetric-key cryptography (typically AES or triple DES), although public-key cryptography is widely used for the authentication protocols themselves and for establishing the session key.

The point of using a new, randomly chosen session key for each new connection is to minimize the amount of traffic that gets sent with the users' secret keys or public keys, to reduce the amount of ciphertext an intruder can obtain, and to minimize the damage done if a process crashes and its core dump falls into the wrong hands. Hopefully, the only key present then will be the session key. All the permanent keys should have been carefully zeroed out after the session was established.

8.7.1 Authentication Based on a Shared Secret Key

For our first authentication protocol, we will assume that Alice and Bob already share a secret key, K_{AB}. This shared key might have been agreed upon on the telephone or in person, but, in any event, not on the (insecure) network.

This protocol is based on a principle found in many authentication protocols: one party sends a random number to the other, who then transforms it in a special way and returns the result. Such protocols are called **challenge-response** protocols. In this and subsequent authentication protocols, the following notation will be used:

A, B are the identities of Alice and Bob.

R_i's are the challenges, where i identifies the challenger.

K_i's are keys, where i indicates the owner.

K_S is the session key.

The message sequence for our first shared-key authentication protocol is illustrated in Fig. 8-32. In message 1, Alice sends her identity, A, to Bob in a way that Bob understands. Bob, of course, has no way of knowing whether this message came from Alice or from Trudy, so he chooses a challenge, a large random number, R_B, and sends it back to "Alice" as message 2, in plaintext. Alice then encrypts the message with the key she shares with Bob and sends the ciphertext, $K_{AB}(R_B)$, back in message 3. When Bob sees this message, he immediately knows that it came from Alice because Trudy does not know K_{AB} and thus could not have generated it. Furthermore, since R_B was chosen randomly from a large space (say, 128-bit random numbers), it is very unlikely that Trudy would have seen R_B and its response in an earlier session. It is equally unlikely that she could guess the correct response to any challenge.

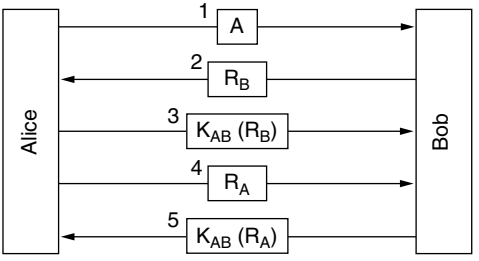


Figure 8-32. Two-way authentication using a challenge-response protocol.

At this point, Bob is sure he is talking to Alice, but Alice is not sure of anything. For all Alice knows, Trudy might have intercepted message 1 and sent back R_B in response. Maybe Bob died last night. To find out to whom she is talking, Alice picks a random number, R_A, and sends it to Bob as plaintext, in message 4. When Bob responds with $K_{AB}(R_A)$, Alice knows she is talking to Bob. If they wish to establish a session key now, Alice can pick one, K_S, and send it to Bob encrypted with K_{AB}.

The protocol of Fig. 8-32 contains five messages. Let us see if we can be clever and eliminate some of them. One approach is illustrated in Fig. 8-33. Here Alice initiates the challenge-response protocol instead of waiting for Bob to do it. Similarly, while he is responding to Alice's challenge, Bob sends his own. The entire protocol can be reduced to three messages instead of five.

Is this new protocol an improvement over the original one? In one sense it is: it is shorter. Unfortunately, it is also wrong. Under certain circumstances, Trudy can defeat this protocol by using what is known as a **reflection attack**. In particular, Trudy can break it if it is possible to open multiple sessions with Bob at once. This situation would be true, for example, if Bob is a bank and is prepared to accept many simultaneous connections from teller machines at once.

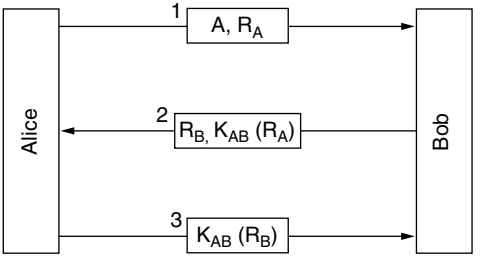


Figure 8-33. A shortened two-way authentication protocol.

Trudy's reflection attack is shown in Fig. 8-34. It starts out with Trudy claiming she is Alice and sending R_T. Bob responds, as usual, with his own challenge, R_B. Now Trudy is stuck. What can she do? She does not know $K_{AB}(R_B)$.

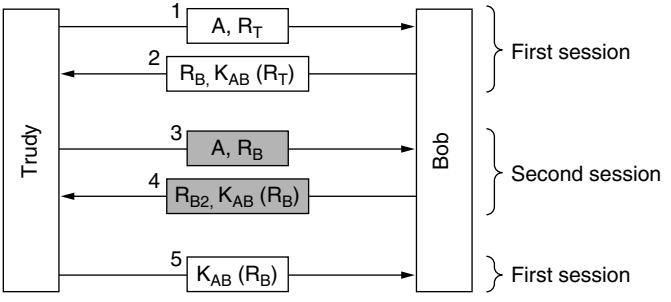


Figure 8-34. The reflection attack.

She can open a second session with message 3, supplying the R_B taken from message 2 as her challenge. Bob calmly encrypts it and sends back $K_{AB}(R_B)$ in message 4. We have shaded the messages on the second session to make them stand out. Now Trudy has the missing information, so she can complete the first session and abort the second one. Bob is now convinced that Trudy is Alice, so when she asks for her bank account balance, he gives it to her without question. Then when she asks him to transfer it all to a secret bank account in Switzerland, he does so without a moment's hesitation.

The moral of this story is:

Designing a correct authentication protocol is much harder than it looks.

The following four general rules often help the designer avoid common pitfalls:

1. Have the initiator prove who she is before the responder has to. This avoids Bob giving away valuable information before Trudy has to give any evidence of who she is.
2. Have the initiator and responder use different keys for proof, even if this means having two shared keys, K_{AB} and K'_{AB}.
3. Have the initiator and responder draw their challenges from different sets. For example, the initiator must use even numbers and the responder must use odd numbers.
4. Make the protocol resistant to attacks involving a second parallel session in which information obtained in one session is used in a different one.

If even one of these rules is violated, the protocol can frequently be broken. Here, all four rules were violated, with disastrous consequences.

Now let us go take a closer look at Fig. 8-32. Surely that protocol is not subject to a reflection attack? Maybe. It is quite subtle. Trudy was able to defeat our protocol by using a reflection attack because it was possible to open a second session with Bob and trick him into answering his own questions. What would happen if Alice were a general-purpose computer that also accepted multiple sessions, rather than a person at a computer? Let us take a look what Trudy can do.

To see how Trudy's attack works, see Fig. 8-35. Alice starts out by announcing her identity in message 1. Trudy intercepts this message and begins her own session with message 2, claiming to be Bob. Again we have shaded the session 2 messages. Alice responds to message 2 by saying in message 3: "You claim to be Bob? Prove it." At this point, Trudy is stuck because she cannot prove she is Bob.

What does Trudy do now? She goes back to the first session, where it is her turn to send a challenge, and sends the R_A she got in message 3. Alice kindly responds to it in message 5, thus supplying Trudy with the information she needs to send in message 6 in session 2. At this point, Trudy is basically home free because she has successfully responded to Alice's challenge in session 2. She can now cancel session 1, send over any old number for the rest of session 2, and she will have an authenticated session with Alice in session 2.

But Trudy is nasty, and she really wants to rub it in. Instead, of sending any old number over to complete session 2, she waits until Alice sends message 7, Alice's challenge for session 1. Of course, Trudy does not know how to respond, so she uses the reflection attack again, sending back R_{A2} as message 8. Alice conveniently encrypts R_{A2} in message 9. Trudy now switches back to session 1 and sends Alice the number she wants in message 10, conveniently copied from what Alice sent in message 9. At this point Trudy has two fully authenticated sessions with Alice.

This attack has a somewhat different result than the attack on the three-message protocol that we saw in Fig. 8-34. This time, Trudy has two authenticated

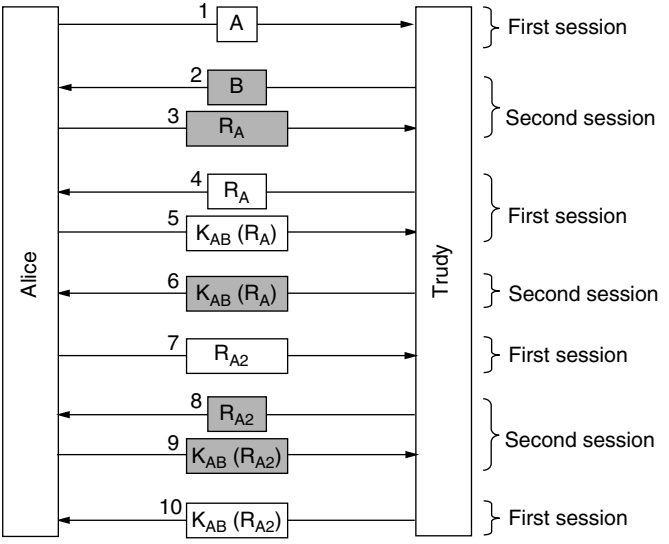


Figure 8-35. A reflection attack on the protocol of Fig. 8-32.

connections with Alice. In the previous example, she had one authenticated connection with Bob. Again here, if we had applied all the general authentication protocol rules discussed earlier, this attack could have been stopped. For a detailed discussion of these kinds of attacks and how to thwart them, see Bird et al. (1993). They also show how it is possible to systematically construct protocols that are provably correct. The simplest such protocol is nevertheless a bit complicated, so we will now show a different class of protocol that also works.

The new authentication protocol is shown in Fig. 8-36 (Bird et al., 1993). It uses an HMAC of the type we saw when studying IPsec. Alice starts out by sending Bob a nonce, R_A, as message 1. Bob responds by selecting his own nonce, R_B, and sending it back along with an HMAC. The HMAC is formed by building a data structure consisting of Alice's nonce, Bob's nonce, their identities, and the shared secret key, K_{AB}. This data structure is then hashed into the HMAC, for example, using SHA-1. When Alice receives message 2, she now has R_A (which she picked herself), R_B, which arrives as plaintext, the two identities, and the secret key, K_{AB}, which she has known all along, so she can compute the HMAC herself. If it agrees with the HMAC in the message, she knows she is talking to Bob because Trudy does not know K_{AB} and thus cannot figure out which HMAC to send. Alice responds to Bob with an HMAC containing just the two nonces.

Can Trudy somehow subvert this protocol? No, because she cannot force either party to encrypt or hash a value of her choice, as happened in Fig. 8-34 and Fig. 8-35. Both HMACs include values chosen by the sending party, something that Trudy cannot control.

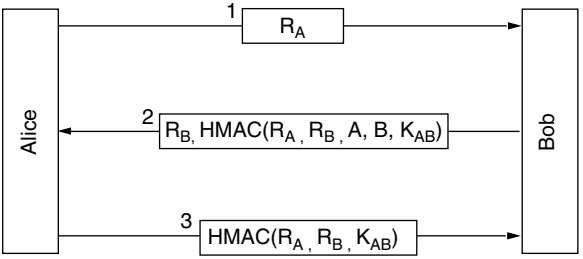


Figure 8-36. Authentication using HMACs.

Using HMACs is not the only way to use this idea. An alternative scheme that is often used instead of computing the HMAC over a series of items is to encrypt the items sequentially using cipher block chaining.

8.7.2 Establishing a Shared Key: The Diffie-Hellman Key Exchange

So far, we have assumed that Alice and Bob share a secret key. Suppose that they do not (because so far there is no universally accepted PKI for signing and distributing certificates). How can they establish one? One way would be for Alice to call Bob and give him her key on the phone, but he would probably start out by saying: "How do I know you are Alice and not Trudy?" They could try to arrange a meeting, with each one bringing a passport, a driver's license, and three major credit cards, but being busy people, they might not be able to find a mutually acceptable date for months. Fortunately, incredible as it may sound, there is a way for total strangers to establish a shared secret key in broad daylight, even with Trudy carefully recording every message.

The protocol that allows strangers to establish a shared secret key is called the **Diffie-Hellman key exchange** (Diffie and Hellman, 1976) and works as follows. Alice and Bob have to agree on two large numbers, n and g, where n is a prime, $(n - 1)/2$ is also a prime, and certain conditions apply to g. These numbers may be public, so either one of them can just pick n and g and tell the other openly. Now Alice picks a large (say, 1024-bit) number, x, and keeps it secret. Similarly, Bob picks a large secret number, y.

Alice initiates the key exchange protocol by sending Bob a message containing $(n, g, g^x \bmod n)$, as shown in Fig. 8-37. Bob responds by sending Alice a message containing $g^y \bmod n$. Now Alice raises the number Bob sent her to the xth power modulo n to get $(g^y \bmod n)^x \bmod n$. Bob performs a similar operation to get $(g^x \bmod n)^y \bmod n$. By the laws of modular arithmetic, both calculations yield $g^{xy} \bmod n$. Lo and behold, as if by magic, Alice and Bob suddenly share a secret key, $g^{xy} \bmod n$.

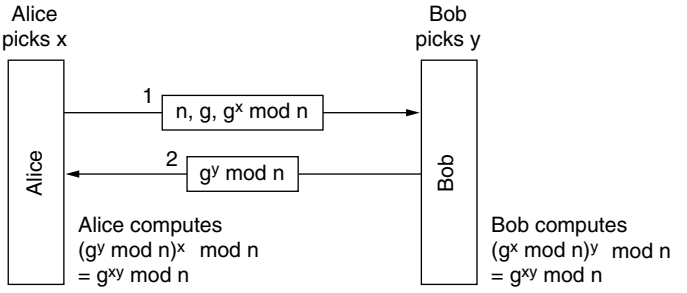


Figure 8-37. The Diffie-Hellman key exchange.

Trudy, of course, has seen both messages. She knows g and n from message 1. If she could compute x and y, she could figure out the secret key. The trouble is, given only $g^x \bmod n$, she cannot find x. No practical algorithm for computing discrete logarithms modulo a very large prime number is known.

To make this example more concrete, we will use the (completely unrealistic) values of $n = 47$ and $g = 3$. Alice picks $x = 8$ and Bob picks $y = 10$. Both of these are kept secret. Alice's message to Bob is (47, 3, 28) because $3^8 \bmod 47$ is 28. Bob's message to Alice is (17). Alice computes $17^8 \bmod 47$, which is 4. Bob computes $28^{10} \bmod 47$, which is 4. Alice and Bob have now independently determined that the secret key is now 4. To find the key, Trudy now has to solve the equation $3^x \bmod 47 = 28$, which can be done by exhaustive search for small numbers like this, but not when all the numbers are hundreds of bits long. All currently known algorithms simply take far too long, even on massively parallel, lightning fast supercomputers.

Despite the elegance of the Diffie-Hellman algorithm, there is a problem: when Bob gets the triple (47, 3, 28), how does he know it is from Alice and not from Trudy? There is no way he can know. Unfortunately, Trudy can exploit this fact to deceive both Alice and Bob, as illustrated in Fig. 8-38. Here, while Alice and Bob are choosing x and y, respectively, Trudy picks her own random number, z. Alice sends message 1, intended for Bob. Trudy intercepts it and sends message 2 to Bob, using the correct g and n (which are public anyway) but with her own z instead of x. She also sends message 3 back to Alice. Later Bob sends message 4 to Alice, which Trudy again intercepts and keeps.

Now everybody does the modular arithmetic. Alice computes the secret key as $g^{xz} \bmod n$, and so does Trudy (for messages to Alice). Bob computes $g^{yz} \bmod n$ and so does Trudy (for messages to Bob). Alice thinks she is talking to Bob, so she establishes a session key (with Trudy). So does Bob. Every message that Alice sends on the encrypted session is captured by Trudy, stored, modified if desired, and then (optionally) passed on to Bob. Similarly, in the other direction, Trudy sees everything and can modify all messages at will, while both Alice and Bob are under the illusion that they have a secure channel to one another. For this

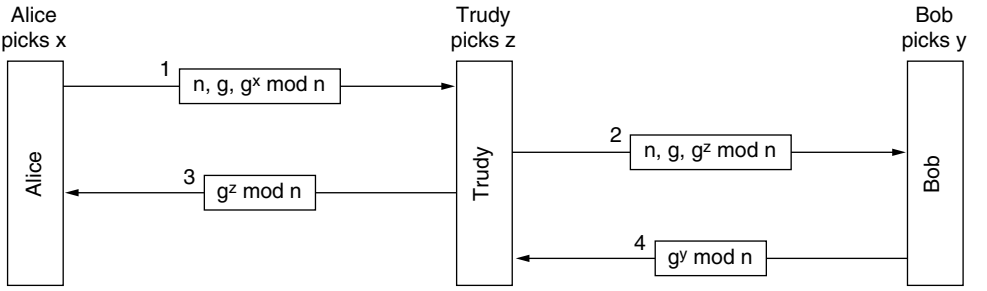


Figure 8-38. The man-in-the-middle attack.

reason, the attack is known as the **man-in-the-middle attack**. It is also called the **bucket brigade attack**, because it vaguely resembles an old-time volunteer fire department passing buckets along the line from the fire truck to the fire.

8.7.3 Authentication Using a Key Distribution Center

Setting up a shared secret with a stranger almost worked, but not quite. On the other hand, it probably was not worth doing in the first place (sour grapes attack). To talk to n people this way, you would need n keys. For popular people, key management would become a real burden, especially if each key had to be stored on a separate plastic chip card.

A different approach is to introduce a trusted key distribution center. In this model, each user has a single key shared with the KDC. Authentication and session key management now go through the KDC. The simplest known KDC authentication protocol involving two parties and a trusted KDC is depicted in Fig. 8-39.

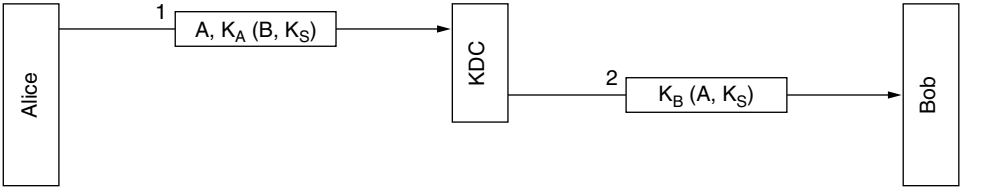


Figure 8-39. A first attempt at an authentication protocol using a KDC.

The idea behind this protocol is simple: Alice picks a session key, K_S, and tells the KDC that she wants to talk to Bob using K_S. This message is encrypted

with the secret key Alice shares (only) with the KDC, K_A. The KDC decrypts this message, extracting Bob's identity and the session key. It then constructs a new message containing Alice's identity and the session key and sends this message to Bob. This encryption is done with K_B, the secret key Bob shares with the KDC. When Bob decrypts the message, he learns that Alice wants to talk to him and which key she wants to use.

The authentication here happens for free. The KDC knows that message 1 must have come from Alice, since no one else would have been able to encrypt it with Alice's secret key. Similarly, Bob knows that message 2 must have come from the KDC, whom he trusts, since no one else knows his secret key.

Unfortunately, this protocol has a serious flaw. Trudy needs some money, so she figures out some legitimate service she can perform for Alice, makes an attractive offer, and gets the job. After doing the work, Trudy then politely requests Alice to pay by bank transfer. Alice then establishes a session key with her banker, Bob. Then she sends Bob a message requesting money to be transferred to Trudy's account.

Meanwhile, Trudy is back to her old ways, snooping on the network. She copies both message 2 in Fig. 8-39 and the money-transfer request that follows it. Later, she replays both of them to Bob who thinks: "Alice must have hired Trudy again. She clearly does good work." Bob then transfers an equal amount of money from Alice's account to Trudy's. Some time after the 50th message pair, Bob runs out of the office to find Trudy to offer her a big loan so she can expand her obviously successful business. This problem is called the **replay attack**.

Several solutions to the replay attack are possible. The first one is to include a timestamp in each message. Then, if anyone receives an obsolete message, it can be discarded. The trouble with this approach is that clocks are never exactly synchronized over a network, so there has to be some interval during which a timestamp is valid. Trudy can replay the message during this interval and get away with it.

The second solution is to put a nonce in each message. Each party then has to remember all previous nonces and reject any message containing a previously used nonce. But nonces have to be remembered forever, lest Trudy try replaying a 5-year-old message. Also, if some machine crashes and it loses its nonce list, it is again vulnerable to a replay attack. Timestamps and nonces can be combined to limit how long nonces have to be remembered, but clearly the protocol is going to get a lot more complicated.

A more sophisticated approach to mutual authentication is to use a multiway challenge-response protocol. A well-known example of such a protocol is the **Needham-Schroeder authentication** protocol (Needham and Schroeder, 1978), one variant of which is shown in Fig. 8-40.

The protocol begins with Alice telling the KDC that she wants to talk to Bob. This message contains a large random number, R_A, as a nonce. The KDC sends back message 2 containing Alice's random number, a session key, and a ticket

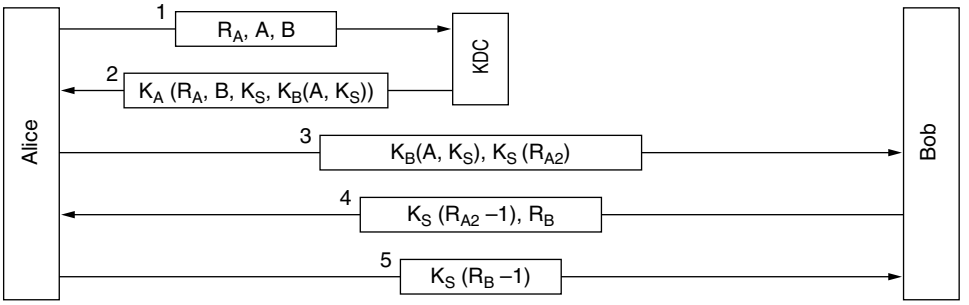


Figure 8-40. The Needham-Schroeder authentication protocol.

that she can send to Bob. The point of the random number, R_A, is to assure Alice that message 2 is fresh, and not a replay. Bob's identity is also enclosed in case Trudy gets any funny ideas about replacing B in message 1 with her own identity so the KDC will encrypt the ticket at the end of message 2 with K_T instead of K_B. The ticket encrypted with K_B is included inside the encrypted message to prevent Trudy from replacing it with something else on the way back to Alice.

Alice now sends the ticket to Bob, along with a new random number, R_{A2}, encrypted with the session key, K_S. In message 4, Bob sends back $K_S(R_{A2} - 1)$ to prove to Alice that she is talking to the real Bob. Sending back $K_S(R_{A2})$ would not have worked, since Trudy could just have stolen it from message 3.

After receiving message 4, Alice is now convinced that she is talking to Bob and that no replays could have been used so far. After all, she just generated R_{A2} a few milliseconds ago. The purpose of message 5 is to convince Bob that it is indeed Alice he is talking to, and no replays are being used here either. By having each party both generate a challenge and respond to one, the possibility of any kind of replay attack is eliminated.

Although this protocol seems pretty solid, it does have a slight weakness. If Trudy ever manages to obtain an old session key in plaintext, she can initiate a new session with Bob by replaying the message 3 that corresponds to the compromised key and convince him that she is Alice (Denning and Sacco, 1981). This time she can plunder Alice's bank account without having to perform the legitimate service even once.

Needham and Schroeder (1987) later published a protocol that corrects this problem. In the same issue of the same journal, Otway and Rees (1987) also published a protocol that solves the problem in a shorter way. Figure 8-41 shows a slightly modified Otway-Rees protocol.

In the Otway-Rees protocol, Alice starts out by generating a pair of random numbers: R, which will be used as a common identifier, and R_A, which Alice will use to challenge Bob. When Bob gets this message, he constructs a new message from the encrypted part of Alice's message and an analogous one of his own.

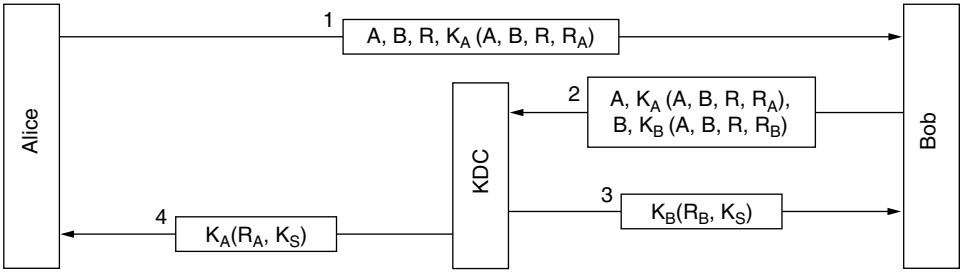


Figure 8-41. The Otway-Rees authentication protocol (slightly simplified).

Both the parts encrypted with K_A and K_B identify Alice and Bob, contain the common identifier, and contain a challenge.

The KDC checks to see if the R in both parts is the same. It might not be if Trudy has tampered with R in message 1 or replaced part of message 2. If the two Rs match, the KDC believes that the request message from Bob is valid. It then generates a session key and encrypts it twice, once for Alice and once for Bob. Each message contains the receiver's random number, as proof that the KDC, and not Trudy, generated the message. At this point, both Alice and Bob are in possession of the same session key and can start communicating. The first time they exchange data messages, each one can see that the other one has an identical copy of K_S, so the authentication is then complete.

8.7.4 Authentication Using Kerberos

An authentication protocol used in many real systems (including Windows 2000 and later versions) is **Kerberos**, which is based on a variant of Needham-Schroeder. It is named for a multiheaded dog in Greek mythology that used to guard the entrance to Hades (presumably to keep undesirables out). Kerberos was designed at M.I.T. to allow workstation users to access network resources in a secure way. Its biggest difference from Needham-Schroeder is its assumption that all clocks are fairly well synchronized. The protocol has gone through several iterations. V5 is the one that is widely used in industry and defined in RFC 4120. The earlier version, V4, was finally retired after serious flaws were found (Yu et al., 2004). V5 improves on V4 with many small changes to the protocol and some improved features, such as the fact that it no longer relies on the now-dated DES. For more information, see Neuman and Ts'o (1994).

Kerberos involves three servers in addition to Alice (a client workstation):

1. Authentication Server (AS): Verifies users during login.
2. Ticket-Granting Server (TGS): Issues "proof of identity tickets."
3. Bob the server: Actually does the work Alice wants performed.

AS is similar to a KDC in that it shares a secret password with every user. The TGS's job is to issue tickets that can convince the real servers that the bearer of a TGS ticket really is who he or she claims to be.

To start a session, Alice sits down at an arbitrary public workstation and types her name. The workstation sends her name and the name of the TGS to the AS in plaintext, as shown in message 1 of Fig. 8-42. What comes back is a session key and a ticket, $K_{TGS}(A, K_S, t)$, intended for the TGS. The session key is encrypted using Alice's secret key, so that only Alice can decrypt it. Only when message 2 arrives does the workstation ask for Alice's password—not before then. The password is then used to generate K_A in order to decrypt message 2 and obtain the session key.

At this point, the workstation overwrites Alice's password to make sure that it is only inside the workstation for a few milliseconds at most. If Trudy tries logging in as Alice, the password she types will be wrong and the workstation will detect this because the standard part of message 2 will be incorrect.

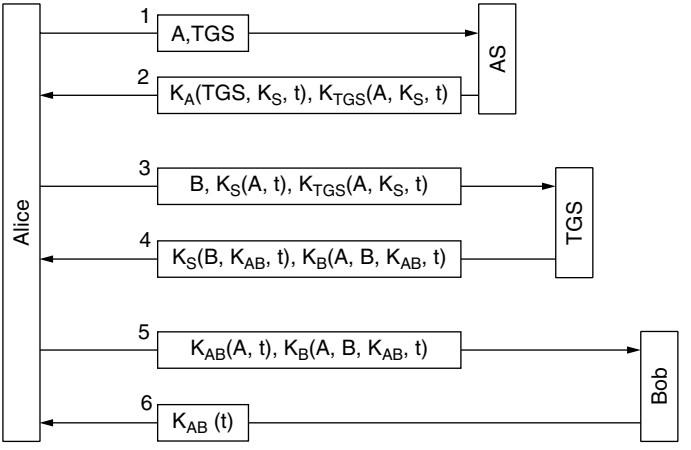


Figure 8-42. The operation of Kerberos V5.

After she logs in, Alice may tell the workstation that she wants to contact Bob the file server. The workstation then sends message 3 to the TGS asking for a ticket to use with Bob. The key element in this request is the ticket $K_{TGS}(A, K_S, t)$, which is encrypted with the TGS's secret key and used as proof that the sender really is Alice. The TGS responds in message 4 by creating a session key, K_{AB}, for Alice to use with Bob. Two versions of it are sent back. The first is encrypted with only K_S, so Alice can read it. The second is another ticket, encrypted with Bob's key, K_B, so Bob can read it.

Trudy can copy message 3 and try to use it again, but she will be foiled by the encrypted timestamp, t, sent along with it. Trudy cannot replace the timestamp with a more recent one, because she does not know K_S, the session key Alice uses to talk to the TGS. Even if Trudy replays message 3 quickly, all she will get is another copy of message 4, which she could not decrypt the first time and will not be able to decrypt the second time either.

Now Alice can send K_{AB} to Bob via the new ticket to establish a session with him (message 5). This exchange is also timestamped. The optional response (message 6) is proof to Alice that she is actually talking to Bob, not to Trudy.

After this series of exchanges, Alice can communicate with Bob under cover of K_{AB}. If she later decides she needs to talk to another server, Carol, she just repeats message 3 to the TGS, only now specifying C instead of B. The TGS will promptly respond with a ticket encrypted with K_C that Alice can send to Carol and that Carol will accept as proof that it came from Alice.

The point of all this work is that now Alice can access servers all over the network in a secure way and her password never has to go over the network. In fact, it only had to be in her own workstation for a few milliseconds. However, note that each server does its own authorization. When Alice presents her ticket to Bob, this merely proves to Bob who sent it. Precisely what Alice is allowed to do is up to Bob.

Since the Kerberos designers did not expect the entire world to trust a single authentication server, they made provision for having multiple **realms**, each with its own AS and TGS. To get a ticket for a server in a distant realm, Alice would ask her own TGS for a ticket accepted by the TGS in the distant realm. If the distant TGS has registered with the local TGS (the same way local servers do), the local TGS will give Alice a ticket valid at the distant TGS. She can then do business over there, such as getting tickets for servers in that realm. Note, however, that for parties in two realms to do business, each one must trust the other's TGS. Otherwise, they cannot do business.

8.7.5 Authentication Using Public-Key Cryptography

Mutual authentication can also be done using public-key cryptography. To start with, Alice needs to get Bob's public key. If a PKI exists with a directory server that hands out certificates for public keys, Alice can ask for Bob's, as shown in Fig. 8-43 as message 1. The reply, in message 2, is an X.509 certificate containing Bob's public key. When Alice verifies that the signature is correct, she sends Bob a message containing her identity and a nonce.

When Bob receives this message, he has no idea whether it came from Alice or from Trudy, but he plays along and asks the directory server for Alice's public key (message 4), which he soon gets (message 5). He then sends Alice message 6, containing Alice's R_A, his own nonce, R_B, and a proposed session key, K_S.

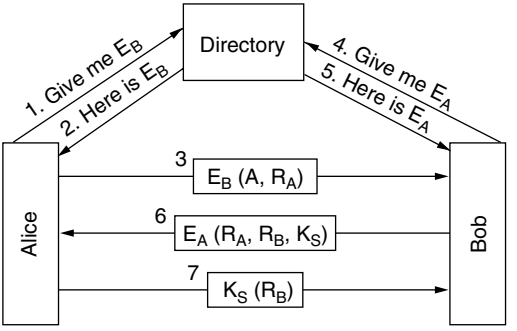


Figure 8-43. Mutual authentication using public-key cryptography.

When Alice gets message 6, she decrypts it using her private key. She sees R_A in it, which gives her a warm feeling inside. The message must have come from Bob, since Trudy has no way of determining R_A. Furthermore, it must be fresh and not a replay, since she just sent Bob R_A. Alice agrees to the session by sending back message 7. When Bob sees R_B encrypted with the session key he just generated, he knows Alice got message 6 and verified R_A. Bob is now a happy camper.

What can Trudy do to try to subvert this protocol? She can fabricate message 3 and trick Bob into probing Alice, but Alice will see an R_A that she did not send and will not proceed further. Trudy cannot forge message 7 back to Bob because she does not know R_B or K_S and cannot determine them without Alice's private key. She is out of luck.

8.8 EMAIL SECURITY

When an email message is sent between two distant sites, it will generally transit dozens of machines on the way. Any of these can read and record the message for future use. In practice, privacy is nonexistent, despite what many people think. Nevertheless, many people would like to be able to send email that can be read by the intended recipient and no one else: not their boss and not even their government. This desire has stimulated several people and groups to apply the cryptographic principles we studied earlier to email to produce secure email. In the following sections we will study a widely used secure email system, PGP, and then briefly mention one other, S/MIME. For additional information about secure email, see Kaufman et al. (2002) and Schneier (1995).

8.8.1 PGP—Pretty Good Privacy

Our first example, **PGP** (**Pretty Good Privacy**) is essentially the brainchild of one person, Phil Zimmermann (1995a, 1995b). Zimmermann is a privacy advocate whose motto is: "If privacy is outlawed, only outlaws will have privacy." Released in 1991, PGP is a complete email security package that provides privacy, authentication, digital signatures, and compression, all in an easy-to-use form. Furthermore, the complete package, including all the source code, is distributed free of charge via the Internet. Due to its quality, price (zero), and easy availability on UNIX, Linux, Windows, and Mac OS platforms, it is widely used today.

PGP encrypts data by using a block cipher called **IDEA** (**International Data Encryption Algorithm**), which uses 128-bit keys. It was devised in Switzerland at a time when DES was seen as tainted and AES had not yet been invented. Conceptually, IDEA is similar to DES and AES: it mixes up the bits in a series of rounds, but the details of the mixing functions are different from DES and AES. Key management uses RSA and data integrity uses MD5, topics that we have already discussed.

PGP has also been embroiled in controversy since day 1 (Levy, 1993). Because Zimmermann did nothing to stop other people from placing PGP on the Internet, where people all over the world could get it, the U.S. Government claimed that Zimmermann had violated U.S. laws prohibiting the export of munitions. The U.S. Government's investigation of Zimmermann went on for 5 years but was eventually dropped, probably for two reasons. First, Zimmermann did not place PGP on the Internet himself, so his lawyer claimed that *he* never exported anything (and then there is the little matter of whether creating a Web site constitutes export at all). Second, the government eventually came to realize that winning a trial meant convincing a jury that a Web site containing a downloadable privacy program was covered by the arms-trafficking law prohibiting the export of war materiel such as tanks, submarines, military aircraft, and nuclear weapons. Years of negative publicity probably did not help much, either.

As an aside, the export rules are bizarre, to put it mildly. The government considered putting code on a Web site to be an illegal export and harassed Zimmermann about it for 5 years. On the other hand, when someone published the complete PGP source code, in C, as a book (in a large font with a checksum on each page to make scanning it in easy) and then exported the book, that was fine with the government because books are not classified as munitions. The sword is mightier than the pen, at least for Uncle Sam.

Another problem PGP ran into involved patent infringement. The company holding the RSA patent, RSA Security, Inc., alleged that PGP's use of the RSA algorithm infringed on its patent, but that problem was settled with releases starting at 2.6. Furthermore, PGP uses another patented encryption algorithm, IDEA, whose use caused some problems at first.

Since PGP is open source, various people and groups have modified it and produced a number of versions. Some of these were designed to get around the munitions laws, others were focused on avoiding the use of patented algorithms, and still others wanted to turn it into a closed-source commercial product. Although the munitions laws have now been slightly liberalized (otherwise, products using AES would not have been exportable from the U.S.), and the RSA patent expired in September 2000, the legacy of all these problems is that several incompatible versions of PGP are in circulation, under various names. The discussion below focuses on classic PGP, which is the oldest and simplest version. Another popular version, Open PGP, is described in RFC 2440. Yet another is the GNU Privacy Guard.

PGP intentionally uses existing cryptographic algorithms rather than inventing new ones. It is largely based on algorithms that have withstood extensive peer review and were not designed or influenced by any government agency trying to weaken them. For people who distrust government, this property is a big plus.

PGP supports text compression, secrecy, and digital signatures and also provides extensive key management facilities, but, oddly enough, not email facilities. It is like a preprocessor that takes plaintext as input and produces signed ciphertext in base64 as output. This output can then be emailed, of course. Some PGP implementations call a user agent as the final step to actually send the message.

To see how PGP works, let us consider the example of Fig. 8-44. Here, Alice wants to send a signed plaintext message, P, to Bob in a secure way. Both Alice and Bob have private (D_X) and public (E_X) RSA keys. Let us assume that each one knows the other's public key; we will cover PGP key management shortly.

Alice starts out by invoking the PGP program on her computer. PGP first hashes her message, P, using MD5, and then encrypts the resulting hash using her private RSA key, D_A. When Bob eventually gets the message, he can decrypt the hash with Alice's public key and verify that the hash is correct. Even if someone else (e.g., Trudy) could acquire the hash at this stage and decrypt it with Alice's known public key, the strength of MD5 guarantees that it would be computationally infeasible to produce another message with the same MD5 hash.

The encrypted hash and the original message are now concatenated into a single message, *P1*, and compressed using the ZIP program, which uses the Ziv-Lempel algorithm (Ziv and Lempel, 1977). Call the output of this step *P1.Z*.

Next, PGP prompts Alice for some random input. Both the content and the typing speed are used to generate a 128-bit IDEA message key, K_M (called a session key in the PGP literature, but this is really a misnomer since there is no session). K_M is now used to encrypt *P1.Z* with IDEA in cipher feedback mode. In addition, K_M is encrypted with Bob's public key, E_B. These two components are then concatenated and converted to base64, as we discussed in the section on MIME in Chap. 7. The resulting message contains only letters, digits, and the symbols +, /, and =, which means it can be put into an RFC 822 body and be expected to arrive unmodified.

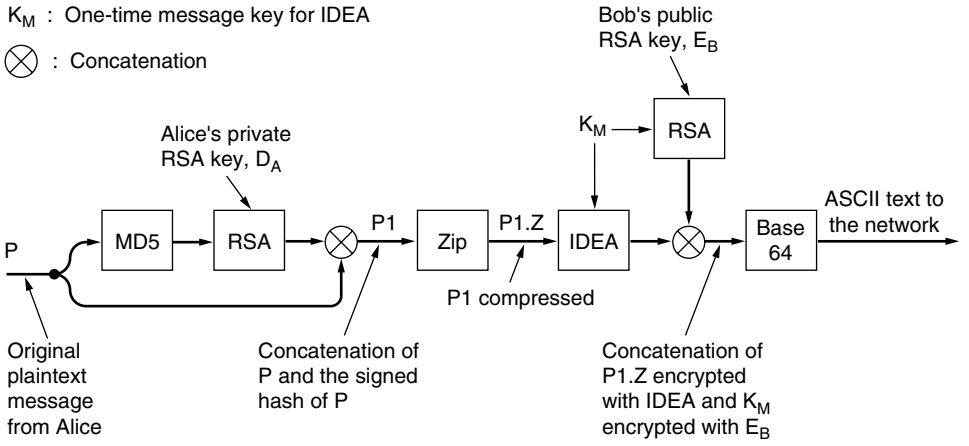


Figure 8-44. PGP in operation for sending a message.

When Bob gets the message, he reverses the base64 encoding and decrypts the IDEA key using his private RSA key. Using this key, he decrypts the message to get *P1.Z*. After decompressing it, Bob separates the plaintext from the encrypted hash and decrypts the hash using Alice's public key. If the plaintext hash agrees with his own MD5 computation, he knows that *P* is the correct message and that it came from Alice.

It is worth noting that RSA is only used in two places here: to encrypt the 128-bit MD5 hash and to encrypt the 128-bit IDEA key. Although RSA is slow, it has to encrypt only 256 bits, not a large volume of data. Furthermore, all 256 plaintext bits are exceedingly random, so a considerable amount of work will be required on Trudy's part just to determine if a guessed key is correct. The heavy-duty encryption is done by IDEA, which is orders of magnitude faster than RSA. Thus, PGP provides security, compression, and a digital signature and does so in a much more efficient way than the scheme illustrated in Fig. 8-19.

PGP supports four RSA key lengths. It is up to the user to select the one that is most appropriate. The lengths are:

1. Casual (384 bits): Can be broken easily today.
2. Commercial (512 bits): Breakable by three-letter organizations.
3. Military (1024 bits): Not breakable by anyone on earth.
4. Alien (2048 bits): Not breakable by anyone on other planets, either.

Since RSA is only used for two small computations, everyone should use alien-strength keys all the time.

The format of a classic PGP message is shown in Fig. 8-45. Numerous other formats are also in use. The message has three parts, containing the IDEA key, the signature, and the message, respectively. The key part contains not only the key, but also a key identifier, since users are permitted to have multiple public keys.

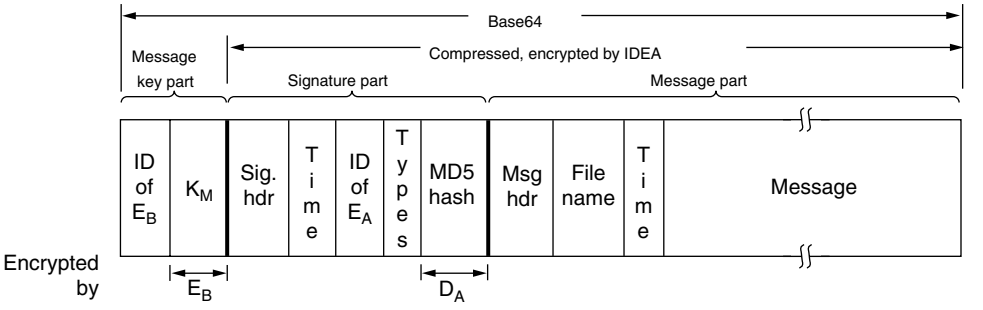


Figure 8-45. A PGP message.

The signature part contains a header, which will not concern us here. The header is followed by a timestamp, the identifier for the sender's public key that can be used to decrypt the signature hash, some type information that identifies the algorithms used (to allow MD6 and RSA2 to be used when they are invented), and the encrypted hash itself.

The message part also contains a header, the default name of the file to be used if the receiver writes the file to the disk, a message creation timestamp, and, finally, the message itself.

Key management has received a large amount of attention in PGP as it is the Achilles' heel of all security systems. Key management works as follows. Each user maintains two data structures locally: a private key ring and a public key ring. The **private key ring** contains one or more personal private/public key pairs. The reason for supporting multiple pairs per user is to permit users to change their public keys periodically or when one is thought to have been compromised, without invalidating messages currently in preparation or in transit. Each pair has an identifier associated with it so that a message sender can tell the recipient which public key was used to encrypt it. Message identifiers consist of the low-order 64 bits of the public key. Users are themselves responsible for avoiding conflicts in their public-key identifiers. The private keys on disk are encrypted using a special (arbitrarily long) password to protect them against sneak attacks.

The **public key ring** contains public keys of the user's correspondents. These are needed to encrypt the message keys associated with each message. Each entry

on the public key ring contains not only the public key, but also its 64-bit identifier and an indication of how strongly the user trusts the key.

The problem being tackled here is the following. Suppose that public keys are maintained on bulletin boards. One way for Trudy to read Bob's secret email is to attack the bulletin board and replace Bob's public key with one of her choice. When Alice later fetches the key allegedly belonging to Bob, Trudy can mount a bucket brigade attack on Bob.

To prevent such attacks, or at least minimize the consequences of them, Alice needs to know how much to trust the item called "Bob's key" on her public key ring. If she knows that Bob personally handed her a CD-ROM containing the key, she can set the trust value to the highest value. It is this decentralized, user-controlled approach to public-key management that sets PGP apart from centralized PKI schemes.

Nevertheless, people do sometimes obtain public keys by querying a trusted key server. For this reason, after X.509 was standardized, PGP supported these certificates as well as the traditional PGP public key ring mechanism. All current versions of PGP have X.509 support.

8.8.2 S/MIME

IETF's venture into email security, called **S/MIME** (**Secure/MIME**), is described in RFCs 2632 through 2643. It provides authentication, data integrity, secrecy, and nonrepudiation. It also is quite flexible, supporting a variety of cryptographic algorithms. Not surprisingly, given the name, S/MIME integrates well with MIME, allowing all kinds of messages to be protected. A variety of new MIME headers are defined, for example, for holding digital signatures.

S/MIME does not have a rigid certificate hierarchy beginning at a single root, which had been one of the political problems that doomed an earlier system called PEM (Privacy Enhanced Mail). Instead, users can have multiple trust anchors. As long as a certificate can be traced back to some trust anchor the user believes in, it is considered valid. S/MIME uses the standard algorithms and protocols we have been examining so far, so we will not discuss it any further here. For the details, please consult the RFCs.

8.9 WEB SECURITY

We have just studied two important areas where security is needed: communications and email. You can think of these as the soup and appetizer. Now it is time for the main course: Web security. The Web is where most of the Trudies hang out nowadays and do their dirty work. In the following sections, we will look at some of the problems and issues relating to Web security.

Web security can be roughly divided into three parts. First, how are objects and resources named securely? Second, how can secure, authenticated connections be established? Third, what happens when a Web site sends a client a piece of executable code? After looking at some threats, we will examine all these issues.

8.9.1 Threats

One reads about Web site security problems in the newspaper almost weekly. The situation is really pretty grim. Let us look at a few examples of what has already happened. First, the home pages of numerous organizations have been attacked and replaced by new home pages of the crackers' choosing. (The popular press calls people who break into computers "hackers," but many programmers reserve that term for great programmers. We prefer to call these people "crackers.") Sites that have been cracked include those belonging to Yahoo!, the U.S. Army, the CIA, NASA, and the *New York Times*. In most cases, the crackers just put up some funny text and the sites were repaired within a few hours.

Now let us look at some much more serious cases. Numerous sites have been brought down by denial-of-service attacks, in which the cracker floods the site with traffic, rendering it unable to respond to legitimate queries. Often, the attack is mounted from a large number of machines that the cracker has already broken into (DDoS attacks). These attacks are so common that they do not even make the news any more, but they can cost the attacked sites thousands of dollars in lost business.

In 1999, a Swedish cracker broke into Microsoft's Hotmail Web site and created a mirror site that allowed anyone to type in the name of a Hotmail user and then read all of the person's current and archived email.

In another case, a 19-year-old Russian cracker named Maxim broke into an e-commerce Web site and stole 300,000 credit card numbers. Then he approached the site owners and told them that if they did not pay him $100,000, he would post all the credit card numbers to the Internet. They did not give in to his blackmail, and he indeed posted the credit card numbers, inflicting great damage on many innocent victims.

In a different vein, a 23-year-old California student emailed a press release to a news agency falsely stating that the Emulex Corporation was going to post a large quarterly loss and that the C.E.O. was resigning immediately. Within hours, the company's stock dropped by 60%, causing stockholders to lose over $2 billion. The perpetrator made a quarter of a million dollars by selling the stock short just before sending the announcement. While this event was not a Web site break-in, it is clear that putting such an announcement on the home page of any big corporation would have a similar effect.

We could (unfortunately) go on like this for many more pages. But it is now time to examine some of the technical issues related to Web security. For more

information about security problems of all kinds, see Anderson (2008a); Stuttard and Pinto (2007); and Schneier (2004). Searching the Internet will also turn up vast numbers of specific cases.

8.9.2 Secure Naming

Let us start with something very basic: Alice wants to visit Bob's Web site. She types Bob's URL into her browser and a few seconds later, a Web page appears. But is it Bob's? Maybe yes and maybe no. Trudy might be up to her old tricks again. For example, she might be intercepting all of Alice's outgoing packets and examining them. When she captures an HTTP *GET* request headed to Bob's Web site, she could go to Bob's Web site herself to get the page, modify it as she wishes, and return the fake page to Alice. Alice would be none the wiser. Worse yet, Trudy could slash the prices at Bob's e-store to make his goods look very attractive, thereby tricking Alice into sending her credit card number to "Bob" to buy some merchandise.

One disadvantage of this classic man-in-the-middle attack is that Trudy has to be in a position to intercept Alice's outgoing traffic and forge her incoming traffic. In practice, she has to tap either Alice's phone line or Bob's, since tapping the fiber backbone is fairly difficult. While active wiretapping is certainly possible, it is a fair amount of work, and while Trudy is clever, she is also lazy. Besides, there are easier ways to trick Alice.

DNS Spoofing

One way would be for Trudy to crack the DNS system or maybe just the DNS cache at Alice's ISP, and replace Bob's IP address (say, 36.1.2.3) with her (Trudy's) IP address (say, 42.9.9.9). That leads to the following attack. The way it is supposed to work is illustrated in Fig. 8-46(a). Here, Alice (1) asks DNS for Bob's IP address, (2) gets it, (3) asks Bob for his home page, and (4) gets that, too. After Trudy has modified Bob's DNS record to contain her own IP address instead of Bob's, we get the situation in Fig. 8-46(b). Here, when Alice looks up Bob's IP address, she gets Trudy's, so all her traffic intended for Bob goes to Trudy. Trudy can now mount a man-in-the-middle attack without having to go to the trouble of tapping any phone lines. Instead, she has to break into a DNS server and change one record, a much easier proposition.

How might Trudy fool DNS? It turns out to be relatively easy. Briefly summarized, Trudy can trick the DNS server at Alice's ISP into sending out a query to look up Bob's address. Unfortunately, since DNS uses UDP, the DNS server has no real way of checking who supplied the answer. Trudy can exploit this property by forging the expected reply and thus injecting a false IP address into the DNS server's cache. For simplicity, we will assume that Alice's ISP does not initially have an entry for Bob's Web site, *bob.com*. If it does, Trudy can wait until it times out and try later (or use other tricks).

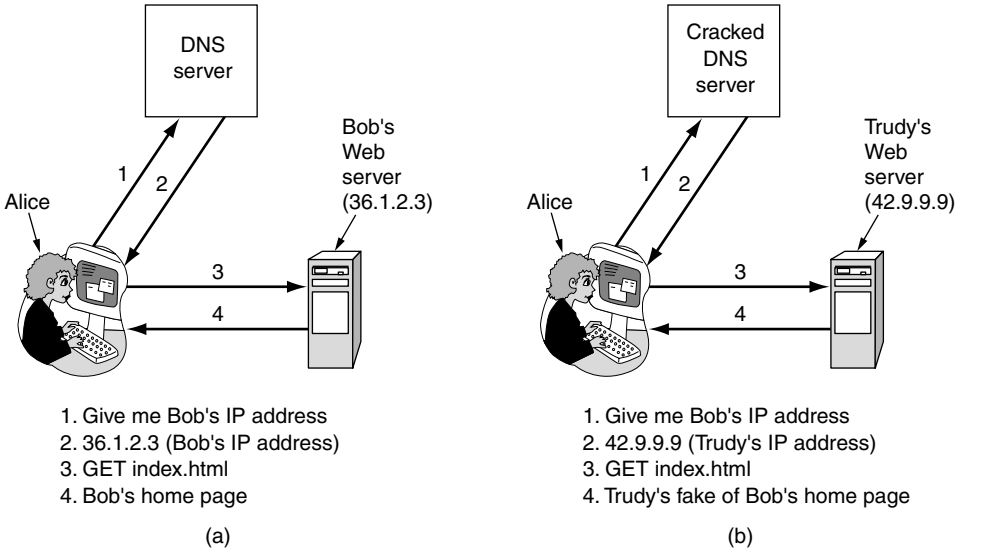


Figure 8-46. (a) Normal situation. (b) An attack based on breaking into a DNS server and modifying Bob's record.

Trudy starts the attack by sending a lookup request to Alice's ISP asking for the IP address of *bob.com*. Since there is no entry for this DNS name, the cache server queries the top-level server for the *com* domain to get one. However, Trudy beats the *com* server to the punch and sends back a false reply saying: "*bob.com* is 42.9.9.9," where that IP address is hers. If her false reply gets back to Alice's ISP first, that one will be cached and the real reply will be rejected as an unsolicited reply to a query no longer outstanding. Tricking a DNS server into installing a false IP address is called **DNS spoofing**. A cache that holds an intentionally false IP address like this is called a **poisoned cache**.

Actually, things are not quite that simple. First, Alice's ISP checks to see that the reply bears the correct IP source address of the top-level server. But since Trudy can put anything she wants in that IP field, she can defeat that test easily since the IP addresses of the top-level servers have to be public.

Second, to allow DNS servers to tell which reply goes with which request, all requests carry a sequence number. To spoof Alice's ISP, Trudy has to know its current sequence number. The easiest way to learn the current sequence number is for Trudy to register a domain herself, say, *trudy-the-intruder.com*. Let us assume its IP address is also 42.9.9.9. She also creates a DNS server for her newly hatched domain, *dns.trudy-the-intruder.com*. It, too, uses Trudy's 42.9.9.9 IP address, since Trudy has only one computer. Now she has to make Alice's ISP aware of her DNS server. That is easy to do. All she has to do is ask Alice's ISP for *foobar.trudy-the-intruder.com*, which will cause Alice's ISP to find out who serves Trudy's new domain by asking the top-level *com* server.

With *dns.trudy-the-intruder.com* safely in the cache at Alice's ISP, the real attack can start. Trudy now queries Alice's ISP for *www.trudy-the-intruder.com*. The ISP naturally sends Trudy's DNS server a query asking for it. This query bears the sequence number that Trudy is looking for. Quick like a bunny, Trudy asks Alice's ISP to look up Bob. She immediately answers her own question by sending the ISP a forged reply, allegedly from the top-level *com* server, saying: "*bob.com* is 42.9.9.9". This forged reply carries a sequence number one higher than the one she just received. While she is at it, she can also send a second forgery with a sequence number two higher, and maybe a dozen more with increasing sequence numbers. One of them is bound to match. The rest will just be thrown out. When Alice's forged reply arrives, it is cached; when the real reply comes in later, it is rejected since no query is then outstanding.

Now when Alice looks up *bob.com*, she is told to use 42.9.9.9, Trudy's address. Trudy has mounted a successful man-in-the-middle attack from the comfort of her own living room. The various steps to this attack are illustrated in Fig. 8-47. This one specific attack can be foiled by having DNS servers use random IDs in their queries rather than just counting, but it seems that every time one hole is plugged, another one turns up. In particular, the IDs are only 16 bits, so working through all of them is easy when it is a computer that is doing the guessing.

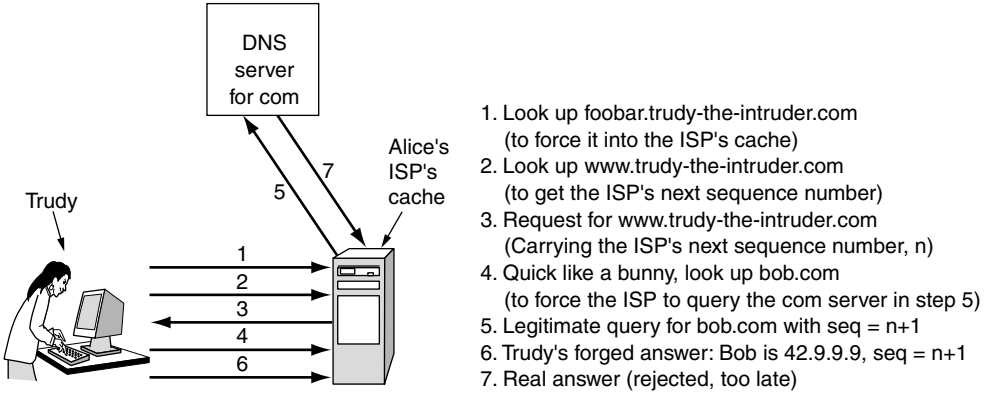


Figure 8-47. How Trudy spoofs Alice's ISP.

Secure DNS

The real problem is that DNS was designed at a time when the Internet was a research facility for a few hundred universities, and neither Alice, nor Bob, nor Trudy was invited to the party. Security was not an issue then; making the Internet work at all was the issue. The environment has changed radically over the

years, so in 1994 IETF set up a working group to make DNS fundamentally secure. This (ongoing) project is known as **DNSsec** (**DNS security**); its first output was presented in RFC 2535. Unfortunately, DNSsec has not been fully deployed yet, so numerous DNS servers are still vulnerable to spoofing attacks.

DNSsec is conceptually extremely simple. It is based on public-key cryptography. Every DNS zone (in the sense of Fig. 7-5) has a public/private key pair. All information sent by a DNS server is signed with the originating zone's private key, so the receiver can verify its authenticity.

DNSsec offers three fundamental services:

1. Proof of where the data originated.
2. Public key distribution.
3. Transaction and request authentication.

The main service is the first one, which verifies that the data being returned has been approved by the zone's owner. The second one is useful for storing and retrieving public keys securely. The third one is needed to guard against playback and spoofing attacks. Note that secrecy is not an offered service since all the information in DNS is considered public. Since phasing in DNSsec is expected to take several years, the ability for security-aware servers to interwork with security-ignorant servers is essential, which implies that the protocol cannot be changed. Let us now look at some of the details.

DNS records are grouped into sets called **RRSets** (**Resource Record Sets**), with all the records having the same name, class, and type being lumped together in a set. An RRSet may contain multiple *A* records, for example, if a DNS name resolves to a primary IP address and a secondary IP address. The RRSets are extended with several new record types (discussed below). Each RRSet is cryptographically hashed (e.g., using SHA-1). The hash is signed by the zone's private key (e.g., using RSA). The unit of transmission to clients is the signed RRSet. Upon receipt of a signed RRSet, the client can verify whether it was signed by the private key of the originating zone. If the signature agrees, the data are accepted. Since each RRSet contains its own signature, RRSets can be cached anywhere, even at untrustworthy servers, without endangering the security.

DNSsec introduces several new record types. The first of these is the *KEY* record. This records holds the public key of a zone, user, host, or other principal, the cryptographic algorithm used for signing, the protocol used for transmission, and a few other bits. The public key is stored naked. X.509 certificates are not used due to their bulk. The algorithm field holds a 1 for MD5/RSA signatures (the preferred choice), and other values for other combinations. The protocol field can indicate the use of IPsec or other security protocols, if any.

The second new record type is the *SIG* record. It holds the signed hash according to the algorithm specified in the *KEY* record. The signature applies to all the records in the RRSet, including any *KEY* records present, but excluding

itself. It also holds the times when the signature begins its period of validity and when it expires, as well as the signer's name and a few other items.

The DNSsec design is such that a zone's private key can be kept offline. Once or twice a day, the contents of a zone's database can be manually transported (e.g., on CD-ROM) to a disconnected machine on which the private key is located. All the RRSets can be signed there and the *SIG* records thus produced can be conveyed back to the zone's primary server on CD-ROM. In this way, the private key can be stored on a CD-ROM locked in a safe except when it is inserted into the disconnected machine for signing the day's new RRSets. After signing is completed, all copies of the key are erased from memory and the disk and the CD-ROM are returned to the safe. This procedure reduces electronic security to physical security, something people understand how to deal with.

This method of presigning RRSets greatly speeds up the process of answering queries since no cryptography has to be done on the fly. The trade-off is that a large amount of disk space is needed to store all the keys and signatures in the DNS databases. Some records will increase tenfold in size due to the signature.

When a client process gets a signed RRSet, it must apply the originating zone's public key to decrypt the hash, compute the hash itself, and compare the two values. If they agree, the data are considered valid. However, this procedure begs the question of how the client gets the zone's public key. One way is to acquire it from a trusted server, using a secure connection (e.g., using IPsec).

However, in practice, it is expected that clients will be preconfigured with the public keys of all the top-level domains. If Alice now wants to visit Bob's Web site, she can ask DNS for the RRSet of *bob.com*, which will contain his IP address and a *KEY* record containing Bob's public key. This RRSet will be signed by the top-level *com* domain, so Alice can easily verify its validity. An example of what this RRSet might contain is shown in Fig. 8-48.

Domain name	Time to live	Class	Type	Value
bob.com.	86400	IN	A	36.1.2.3
bob.com.	86400	IN	KEY	3682793A7B73F731029CE2737D...
bob.com.	86400	IN	SIG	86947503A8B848F5272E53930C...

Figure 8-48. An example RRSet for *bob.com*. The *KEY* record is Bob's public key. The *SIG* record is the top-level *com* server's signed hash of the *A* and *KEY* records to verify their authenticity.

Now armed with a verified copy of Bob's public key, Alice can ask Bob's DNS server (run by Bob) for the IP address of *www.bob.com*. This RRSet will be signed by Bob's private key, so Alice can verify the signature on the RRSet Bob returns. If Trudy somehow manages to inject a false RRSet into any of the caches, Alice can easily detect its lack of authenticity because the *SIG* record contained in it will be incorrect.

However, DNSsec also provides a cryptographic mechanism to bind a response to a specific query, to prevent the kind of spoof Trudy managed to pull off in Fig. 8-47. This (optional) antispoofing measure adds to the response a hash of the query message signed with the respondent's private key. Since Trudy does not know the private key of the top-level *com* server, she cannot forge a response to a query Alice's ISP sent there. She can certainly get her response back first, but it will be rejected due to its invalid signature over the hashed query.

DNSsec also supports a few other record types. For example, the *CERT* record can be used for storing (e.g., X.509) certificates. This record has been provided because some people want to turn DNS into a PKI. Whether this will actually happen remains to be seen. We will stop our discussion of DNSsec here. For more details, please consult RFC 2535.

8.9.3 SSL—The Secure Sockets Layer

Secure naming is a good start, but there is much more to Web security. The next step is secure connections. We will now look at how secure connections can be achieved. Nothing involving security is simple and this is not either.

When the Web burst into public view, it was initially used for just distributing static pages. However, before long, some companies got the idea of using it for financial transactions, such as purchasing merchandise by credit card, online banking, and electronic stock trading. These applications created a demand for secure connections. In 1995, Netscape Communications Corp., the then-dominant browser vendor, responded by introducing a security package called **SSL** (**Secure Sockets Layer**) to meet this demand. This software and its protocol are now widely used, for example, by Firefox, Safari, and Internet Explorer, so it is worth examining in some detail.

SSL builds a secure connection between two sockets, including

1. Parameter negotiation between client and server.
2. Authentication of the server by the client.
3. Secret communication.
4. Data integrity protection.

We have seen these items before, so there is no need to elaborate on them.

The positioning of SSL in the usual protocol stack is illustrated in Fig. 8-49. Effectively, it is a new layer interposed between the application layer and the transport layer, accepting requests from the browser and sending them down to TCP for transmission to the server. Once the secure connection has been established, SSL's main job is handling compression and encryption. When HTTP is used over SSL, it is called **HTTPS** (**Secure HTTP**), even though it is the standard HTTP protocol. Sometimes it is available at a new port (443) instead of port 80.

As an aside, SSL is not restricted to Web browsers, but that is its most common application. It can also provide mutual authentication.

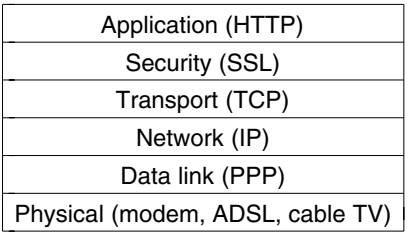


Figure 8-49. Layers (and protocols) for a home user browsing with SSL.

The SSL protocol has gone through several versions. Below we will discuss only version 3, which is the most widely used version. SSL supports a variety of different options. These options include the presence or absence of compression, the cryptographic algorithms to be used, and some matters relating to export restrictions on cryptography. The last is mainly intended to make sure that serious cryptography is used only when both ends of the connection are in the United States. In other cases, keys are limited to 40 bits, which cryptographers regard as something of a joke. Netscape was forced to put in this restriction in order to get an export license from the U.S. Government.

SSL consists of two subprotocols, one for establishing a secure connection and one for using it. Let us start out by seeing how secure connections are established. The connection establishment subprotocol is shown in Fig. 8-50. It starts out with message 1 when Alice sends a request to Bob to establish a connection. The request specifies the SSL version Alice has and her preferences with respect to compression and cryptographic algorithms. It also contains a nonce, R_A, to be used later.

Now it is Bob's turn. In message 2, Bob makes a choice among the various algorithms that Alice can support and sends his own nonce, R_B. Then, in message 3, he sends a certificate containing his public key. If this certificate is not signed by some well-known authority, he also sends a chain of certificates that can be followed back to one. All browsers, including Alice's, come preloaded with about 100 public keys, so if Bob can establish a chain anchored to one of these, Alice will be able to verify Bob's public key. At this point, Bob may send some other messages (such as a request for Alice's public-key certificate). When Bob is done, he sends message 4 to tell Alice it is her turn.

Alice responds by choosing a random 384-bit **premaster key** and sending it to Bob encrypted with his public key (message 5). The actual session key used for encrypting data is derived from the premaster key combined with both nonces in a complex way. After message 5 has been received, both Alice and Bob are able to compute the session key. For this reason, Alice tells Bob to switch to the

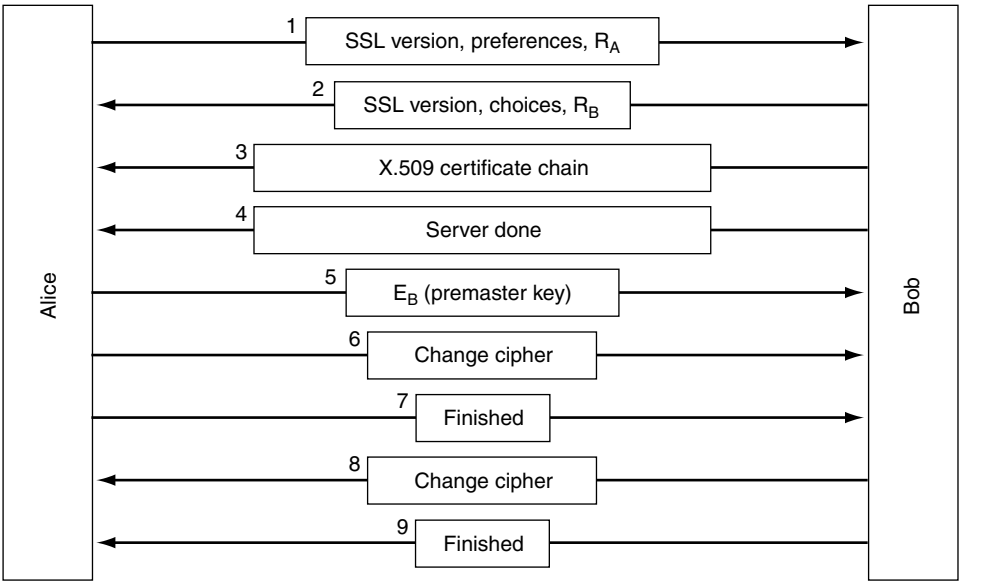


Figure 8-50. A simplified version of the SSL connection establishment subprotocol.

new cipher (message 6) and also that she is finished with the establishment subprotocol (message 7). Bob then acknowledges her (messages 8 and 9).

However, although Alice knows who Bob is, Bob does not know who Alice is (unless Alice has a public key and a corresponding certificate for it, an unlikely situation for an individual). Therefore, Bob's first message may well be a request for Alice to log in using a previously established login name and password. The login protocol, however, is outside the scope of SSL. Once it has been accomplished, by whatever means, data transport can begin.

As mentioned above, SSL supports multiple cryptographic algorithms. The strongest one uses triple DES with three separate keys for encryption and SHA-1 for message integrity. This combination is relatively slow, so it is mostly used for banking and other applications in which the highest security is required. For ordinary e-commerce applications, RC4 is used with a 128-bit key for encryption and MD5 is used for message authentication. RC4 takes the 128-bit key as a seed and expands it to a much larger number for internal use. Then it uses this internal number to generate a keystream. The keystream is XORed with the plaintext to provide a classical stream cipher, as we saw in Fig. 8-14. The export versions also use RC4 with 128-bit keys, but 88 of the bits are made public to make the cipher easy to break.

For actual transport, a second subprotocol is used, as shown in Fig. 8-51. Messages from the browser are first broken into units of up to 16 KB. If data

compression is enabled, each unit is then separately compressed. After that, a secret key derived from the two nonces and premaster key is concatenated with the compressed text and the result is hashed with the agreed-on hashing algorithm (usually MD5). This hash is appended to each fragment as the MAC. The compressed fragment plus MAC is then encrypted with the agreed-on symmetric encryption algorithm (usually by XORing it with the RC4 keystream). Finally, a fragment header is attached and the fragment is transmitted over the TCP connection.

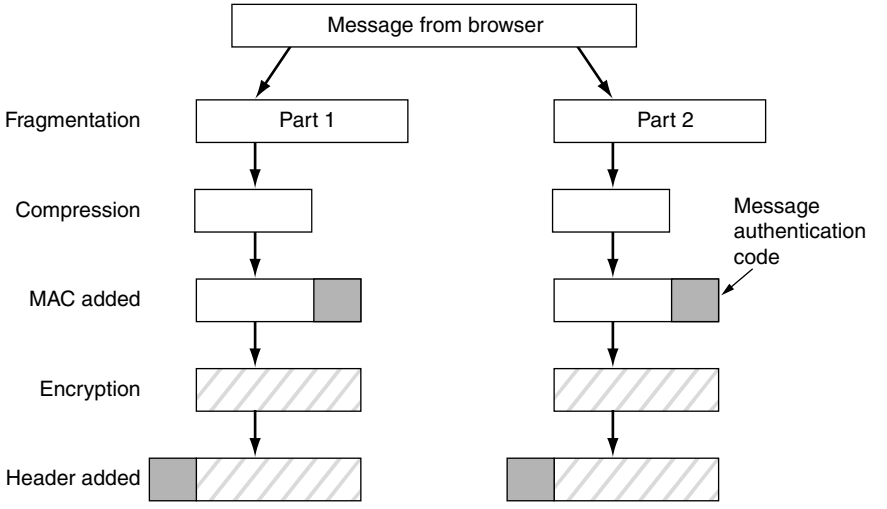


Figure 8-51. Data transmission using SSL.

A word of caution is in order, however. Since it has been shown that RC4 has some weak keys that can be easily cryptanalyzed, the security of SSL using RC4 is on shaky ground (Fluhrer et al., 2001). Browsers that allow the user to choose the cipher suite should be configured to use triple DES with 168-bit keys and SHA-1 all the time, even though this combination is slower than RC4 and MD5. Or, better yet, users should upgrade to browsers that support the successor to SSL that we describe shortly.

A problem with SSL is that the principals may not have certificates, and even if they do, they do not always verify that the keys being used match them.

In 1996, Netscape Communications Corp. turned SSL over to IETF for standardization. The result was **TLS** (**Transport Layer Security**). It is described in RFC 5246.

TLS was built on SSL version 3. The changes made to SSL were relatively small, but just enough that SSL version 3 and TLS cannot interoperate. For example, the way the session key is derived from the premaster key and nonces was

changed to make the key stronger (i.e., harder to cryptanalyze). Because of this incompatibility, most browsers implement both protocols, with TLS falling back to SSL during negotiation if necessary. This is referred to as SSL/TLS. The first TLS implementation appeared in 1999 with version 1.2 defined in August 2008. It includes support for stronger cipher suites (notably AES). SSL has remained strong in the marketplace although TLS will probably gradually replace it.

8.9.4 Mobile Code Security

Naming and connections are two areas of concern related to Web security. But there are more. In the early days, when Web pages were just static HTML files, they did not contain executable code. Now they often contain small programs, including Java applets, ActiveX controls, and JavaScripts. Downloading and executing such **mobile code** is obviously a massive security risk, so various methods have been devised to minimize it. We will now take a quick peek at some of the issues raised by mobile code and some approaches to dealing with it.

Java Applet Security

Java applets are small Java programs compiled to a stack-oriented machine language called **JVM** (**Java Virtual Machine**). They can be placed on a Web page for downloading along with the page. After the page is loaded, the applets are inserted into a JVM interpreter inside the browser, as illustrated in Fig. 8-52.

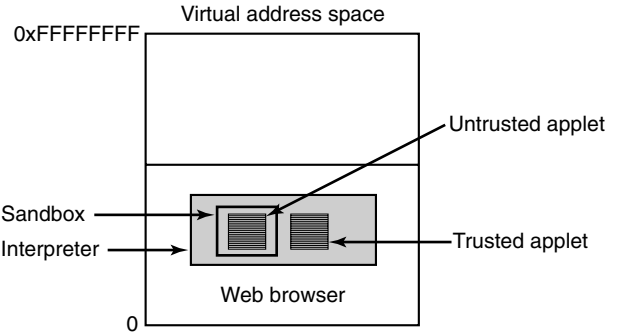


Figure 8-52. Applets can be interpreted by a Web browser.

The advantage of running interpreted code over compiled code is that every instruction is examined by the interpreter before being executed. This gives the interpreter the opportunity to check whether the instruction's address is valid. In addition, system calls are also caught and interpreted. How these calls are handled is a matter of the security policy. For example, if an applet is trusted (e.g., it

came from the local disk), its system calls could be carried out without question. However, if an applet is not trusted (e.g., it came in over the Internet), it could be encapsulated in what is called a **sandbox** to restrict its behavior and trap its attempts to use system resources.

When an applet tries to use a system resource, its call is passed to a security monitor for approval. The monitor examines the call in light of the local security policy and then makes a decision to allow or reject it. In this way, it is possible to give applets access to some resources but not all. Unfortunately, the reality is that the security model works badly and that bugs in it crop up all the time.

ActiveX

ActiveX controls are x86 binary programs that can be embedded in Web pages. When one of them is encountered, a check is made to see if it should be executed, and it if passes the test, it is executed. It is not interpreted or sandboxed in any way, so it has as much power as any other user program and can potentially do great harm. Thus, all the security is in the decision whether to run the ActiveX control. In retrospect, the whole idea is a gigantic security hole.

The method that Microsoft chose for making this decision is based on the idea of **code signing**. Each ActiveX control is accompanied by a digital signature—a hash of the code that is signed by its creator using public-key cryptography. When an ActiveX control shows up, the browser first verifies the signature to make sure it has not been tampered with in transit. If the signature is correct, the browser then checks its internal tables to see if the program's creator is trusted or there is a chain of trust back to a trusted creator. If the creator is trusted, the program is executed; otherwise, it is not. The Microsoft system for verifying ActiveX controls is called **Authenticode**.

It is useful to contrast the Java and ActiveX approaches. With the Java approach, no attempt is made to determine who wrote the applet. Instead, a run-time interpreter makes sure it does not do things the machine owner has said applets may not do. In contrast, with code signing, there is no attempt to monitor the mobile code's run-time behavior. If it came from a trusted source and has not been modified in transit, it just runs. No attempt is made to see whether the code is malicious or not. If the original programmer *intended* the code to format the hard disk and then erase the flash ROM so the computer can never again be booted, and if the programmer has been certified as trusted, the code will be run and destroy the computer (unless ActiveX controls have been disabled in the browser).

Many people feel that trusting an unknown software company is scary. To demonstrate the problem, a programmer in Seattle formed a software company and got it certified as trustworthy, which is easy to do. He then wrote an ActiveX control that did a clean shutdown of the machine and distributed his ActiveX control widely. It shut down many machines, but they could just be rebooted, so no

harm was done. He was just trying to expose the problem to the world. The official response was to revoke the certificate for this specific ActiveX control, which ended a short episode of acute embarrassment, but the underlying problem is still there for an evil programmer to exploit (Garfinkel with Spafford, 2002). Since there is no way to police the thousands of software companies that might write mobile code, the technique of code signing is a disaster waiting to happen.

JavaScript

JavaScript does not have any formal security model, but it does have a long history of leaky implementations. Each vendor handles security in a different way. For example, Netscape Navigator version 2 used something akin to the Java model, but by version 4 that had been abandoned for a code-signing model.

The fundamental problem is that letting foreign code run on your machine is asking for trouble. From a security standpoint, it is like inviting a burglar into your house and then trying to watch him carefully so he cannot escape from the kitchen into the living room. If something unexpected happens and you are distracted for a moment, bad things can happen. The tension here is that mobile code allows flashy graphics and fast interaction, and many Web site designers think that this is much more important than security, especially when it is somebody else's machine at risk.

Browser Extensions

As well as extending Web pages with code, there is a booming marketplace in **browser extensions**, **add-ons**, and **plug-ins**. They are computer programs that extend the functionality of Web browsers. Plug-ins often provide the capability to interpret or display a certain type of content, such as PDFs or Flash animations. Extensions and add-ons provide new browser features, such as better password management, or ways to interact with pages by, for example, marking them up or enabling easy shopping for related items.

Installing an extension, add-on, or plug-in is as simple as coming across something you want when browsing and following the link to install the program. This action will cause code to be downloaded across the Internet and installed into the browser. All of these programs are written to frameworks that differ depending on the browser that is being enhanced. However, to a first approximation, they become part of the trusted computing base of the browser. That is, if the code that is installed is buggy, the entire browser can be compromised.

There are two other obvious failure modes as well. The first is that the program may behave maliciously, for example, by gathering personal information and sending it to a remote server. For all the browser knows, the user installed the extension for precisely this purpose. The second problem is that plug-ins give the browser the ability to interpret new types of content. Often this content is a full

blown programming language itself. PDF and Flash are good examples. When users view pages with PDF and Flash content, the plug-ins in their browser are executing the PDF and Flash code. That code had better be safe; often there are vulnerabilities that it can exploit. For all of these reasons, add-ons and plug-ins should only be installed as needed and only from trusted vendors.

Viruses

Viruses are another form of mobile code. Only, unlike the examples above, viruses are not invited in at all. The difference between a virus and ordinary mobile code is that viruses are written to reproduce themselves. When a virus arrives, either via a Web page, an email attachment, or some other way, it usually starts out by infecting executable programs on the disk. When one of these programs is run, control is transferred to the virus, which usually tries to spread itself to other machines, for example, by emailing copies of itself to everyone in the victim's email address book. Some viruses infect the boot sector of the hard disk, so when the machine is booted, the virus gets to run. Viruses have become a huge problem on the Internet and have caused billions of dollars' worth of damage. There is no obvious solution. Perhaps a whole new generation of operating systems based on secure microkernels and tight compartmentalization of users, processes, and resources might help.

8.10 SOCIAL ISSUES

The Internet and its security technology is an area where social issues, public policy, and technology meet head on, often with huge consequences. Below we will just briefly examine three areas: privacy, freedom of speech, and copyright. Needless to say, we can only scratch the surface. For additional reading, see Anderson (2008a), Garfinkel with Spafford (2002), and Schneier (2004). The Internet is also full of material. Just type words such as "privacy," "censorship," and "copyright" into any search engine. Also, see this book's Web site for some links. It is at *http://www.pearsonhighered.com/tanenbaum*.

8.10.1 Privacy

Do people have a right to privacy? Good question. The Fourth Amendment to the U.S. Constitution prohibits the government from searching people's houses, papers, and effects without good reason, and goes on to restrict the circumstances under which search warrants shall be issued. Thus, privacy has been on the public agenda for over 200 years, at least in the U.S.

What has changed in the past decade is both the ease with which governments can spy on their citizens and the ease with which the citizens can prevent such

spying. In the 18th century, for the government to search a citizen's papers, it had to send out a policeman on a horse to go to the citizen's farm demanding to see certain documents. It was a cumbersome procedure. Nowadays, telephone companies and Internet providers readily provide wiretaps when presented with search warrants. It makes life much easier for the policeman and there is no danger of falling off a horse.

Cryptography changes all that. Anybody who goes to the trouble of downloading and installing PGP and who uses a well-guarded alien-strength key can be fairly sure that nobody in the known universe can read his email, search warrant or no search warrant. Governments well understand this and do not like it. Real privacy means it is much harder for them to spy on criminals of all stripes, but it is also much harder to spy on journalists and political opponents. Consequently, some governments restrict or forbid the use or export of cryptography. In France, for example, prior to 1999, all cryptography was banned unless the government was given the keys.

France was not alone. In April 1993, the U.S. Government announced its intention to make a hardware cryptoprocessor, the **clipper chip**, the standard for all networked communication. It was said that this would guarantee citizens' privacy. It also mentioned that the chip provided the government with the ability to decrypt all traffic via a scheme called **key escrow**, which allowed the government access to all the keys. However, the government promised only to snoop when it had a valid search warrant. Needless to say, a huge furor ensued, with privacy advocates denouncing the whole plan and law enforcement officials praising it. Eventually, the government backed down and dropped the idea.

A large amount of information about electronic privacy is available at the Electronic Frontier Foundation's Web site, *www.eff.org*.

Anonymous Remailers

PGP, SSL, and other technologies make it possible for two parties to establish secure, authenticated communication, free from third-party surveillance and interference. However, sometimes privacy is best served by *not* having authentication, in fact, by making communication anonymous. The anonymity may be desired for point-to-point messages, newsgroups, or both.

Let us consider some examples. First, political dissidents living under authoritarian regimes often wish to communicate anonymously to escape being jailed or killed. Second, wrongdoing in many corporate, educational, governmental, and other organizations has often been exposed by whistleblowers, who frequently prefer to remain anonymous to avoid retribution. Third, people with unpopular social, political, or religious views may wish to communicate with each other via email or newsgroups without exposing themselves. Fourth, people may wish to discuss alcoholism, mental illness, sexual harassment, child abuse, or being a

member of a persecuted minority in a newsgroup without having to go public. Numerous other examples exist, of course.

Let us consider a specific example. In the 1990s, some critics of a nontraditional religious group posted their views to a USENET newsgroup via an **anonymous remailer**. This server allowed users to create pseudonyms and send email to the server, which then remailed or re-posted them using the pseudonyms, so no one could tell where the messages really came from. Some postings revealed what the religious group claimed were trade secrets and copyrighted documents. The religious group responded by telling local authorities that its trade secrets had been disclosed and its copyright infringed, both of which were crimes where the server was located. A court case followed and the server operator was compelled to turn over the mapping information that revealed the true identities of the persons who had made the postings. (Incidentally, this was not the first time that a religious group was unhappy when someone leaked its trade secrets: William Tyndale was burned at the stake in 1536 for translating the Bible into English).

A substantial segment of the Internet community was completely outraged by this breach of confidentiality. The conclusion that everyone drew is that an anonymous remailer that stores a mapping between real email addresses and pseudonyms (now called a type 1 remailer) is not worth much. This case stimulated various people into designing anonymous remailers that could withstand subpoena attacks.

These new remailers, often called **cypherpunk remailers**, work as follows. The user produces an email message, complete with RFC 822 headers (except *From:*, of course), encrypts it with the remailer's public key, and sends it to the remailer. There the outer RFC 822 headers are stripped off, the content is decrypted and the message is remailed. The remailer has no accounts and maintains no logs, so even if the server is later confiscated, it retains no trace of messages that have passed through it.

Many users who wish anonymity chain their requests through multiple anonymous remailers, as shown in Fig. 8-53. Here, Alice wants to send Bob a really, really, really anonymous Valentine's Day card, so she uses three remailers. She composes the message, M, and puts a header on it containing Bob's email address. Then she encrypts the whole thing with remailer 3's public key, E_3 (indicated by horizontal hatching). To this she prepends a header with remailer 3's email address in plaintext. This is the message shown between remailers 2 and 3 in the figure.

Then she encrypts this message with remailer 2's public key, E_2 (indicated by vertical hatching) and prepends a plaintext header containing remailer 2's email address. This message is shown between 1 and 2 in Fig. 8-53. Finally, she encrypts the entire message with remailer 1's public key, E_1, and prepends a plaintext header with remailer 1's email address. This is the message shown to the right of Alice in the figure and this is the message she actually transmits.

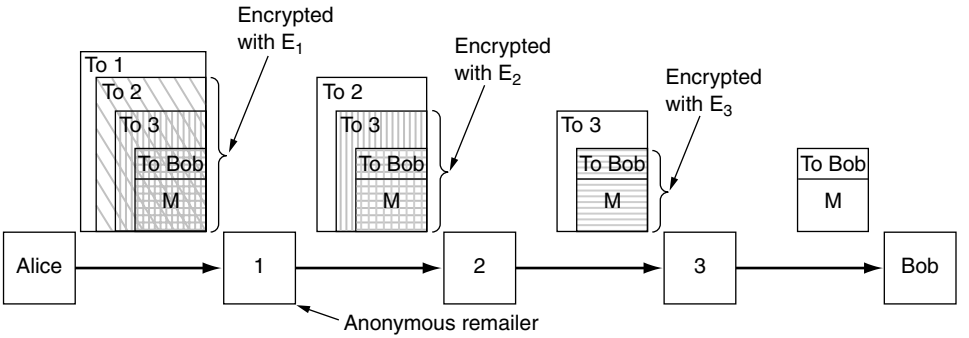


Figure 8-53. How Alice uses three remailers to send Bob a message.

When the message hits remailer 1, the outer header is stripped off. The body is decrypted and then emailed to remailer 2. Similar steps occur at the other two remailers.

Although it is extremely difficult for anyone to trace the final message back to Alice, many remailers take additional safety precautions. For example, they may hold messages for a random time, add or remove junk at the end of a message, and reorder messages, all to make it harder for anyone to tell which message output by a remailer corresponds to which input, in order to thwart traffic analysis. For a description of this kind of remailer, see Mazières and Kaashoek (1998).

Anonymity is not restricted to email. Services also exist that allow anonymous Web surfing using the same form of layered path in which one node only knows the next node in the chain. This method is called **onion routing** because each node peels off another layer of the onion to determine where to forward the packet next. The user configures his browser to use the anonymizer service as a proxy. Tor is a well-known example of such a system (Dingledine et al., 2004). Henceforth, all HTTP requests go through the anonymizer network, which requests the page and sends it back. The Web site sees an exit node of the anonymizer network as the source of the request, not the user. As long as the anonymizer network refrains from keeping a log, after the fact no one can determine who requested which page.

8.10.2 Freedom of Speech

Privacy relates to individuals wanting to restrict what other people can see about them. A second key social issue is freedom of speech, and its opposite, censorship, which is about governments wanting to restrict what individuals can read and publish. With the Web containing millions and millions of pages, it has become a censor's paradise. Depending on the nature and ideology of the regime, banned material may include Web sites containing any of the following:

1. Material inappropriate for children or teenagers.
2. Hate aimed at various ethnic, religious, sexual or other groups.
3. Information about democracy and democratic values.
4. Accounts of historical events contradicting the government's version.
5. Manuals for picking locks, building weapons, encrypting messages, etc.

The usual response is to ban the "bad" sites.

Sometimes the results are unexpected. For example, some public libraries have installed Web filters on their computers to make them child friendly by blocking pornography sites. The filters veto sites on their blacklists but also check pages for dirty words before displaying them. In one case in Loudoun County, Virginia, the filter blocked a patron's search for information on breast cancer because the filter saw the word "breast." The library patron sued Loudoun County. However, in Livermore, California, a parent sued the public library for *not* installing a filter after her 12-year-old son was caught viewing pornography there. What's a library to do?

It has escaped many people that the World Wide Web is a worldwide Web. It covers the whole world. Not all countries agree on what should be allowed on the Web. For example, in November 2000, a French court ordered Yahoo!, a California Corporation, to block French users from viewing auctions of Nazi memorabilia on Yahoo!'s Web site because owning such material violates French law. Yahoo! appealed to a U.S. court, which sided with it, but the issue of whose laws apply where is far from settled.

Just imagine. What would happen if some court in Utah instructed France to block Web sites dealing with wine because they do not comply with Utah's much stricter laws about alcohol? Suppose that China demanded that all Web sites dealing with democracy be banned as not in the interest of the State. Do Iranian laws on religion apply to more liberal Sweden? Can Saudi Arabia block Web sites dealing with women's rights? The whole issue is a veritable Pandora's box.

A relevant comment from John Gilmore is: "The net interprets censorship as damage and routes around it." For a concrete implementation, consider the **eternity service** (Anderson, 1996). Its goal is to make sure published information cannot be depublished or rewritten, as was common in the Soviet Union during Josef Stalin's reign. To use the eternity service, the user specifies how long the material is to be preserved, pays a fee proportional to its duration and size, and uploads it. Thereafter, no one can remove or edit it, not even the uploader.

How could such a service be implemented? The simplest model is to use a peer-to-peer system in which stored documents would be placed on dozens of participating servers, each of which gets a fraction of the fee, and thus an incentive to join the system. The servers should be spread over many legal jurisdictions for maximum resilience. Lists of 10 randomly selected servers would be stored

securely in multiple places, so that if some were compromised, others would still exist. An authority bent on destroying the document could never be sure it had found all copies. The system could also be made self-repairing in the sense that if it became known that some copies had been destroyed, the remaining sites would attempt to find new repositories to replace them.

The eternity service was the first proposal for a censorship-resistant system. Since then, others have been proposed and, in some cases, implemented. Various new features have been added, such as encryption, anonymity, and fault tolerance. Often the files to be stored are broken up into multiple fragments, with each fragment stored on many servers. Some of these systems are Freenet (Clarke et al., 2002), PASIS (Wylie et al., 2000), and Publius (Waldman et al., 2000). Other work is reported by Serjantov (2002).

Increasingly, many countries are trying to regulate the export of intangibles, which often include Web sites, software, scientific papers, email, telephone helpdesks, and more. Even the U.K., which has a centuries-long tradition of freedom of speech, is now seriously considering highly restrictive laws, that would, for example, define technical discussions between a British professor and his foreign Ph.D. student, both located at the University of Cambridge, as regulated export needing a government license (Anderson, 2002). Needless to say, many people consider such a policy to be outrageous.

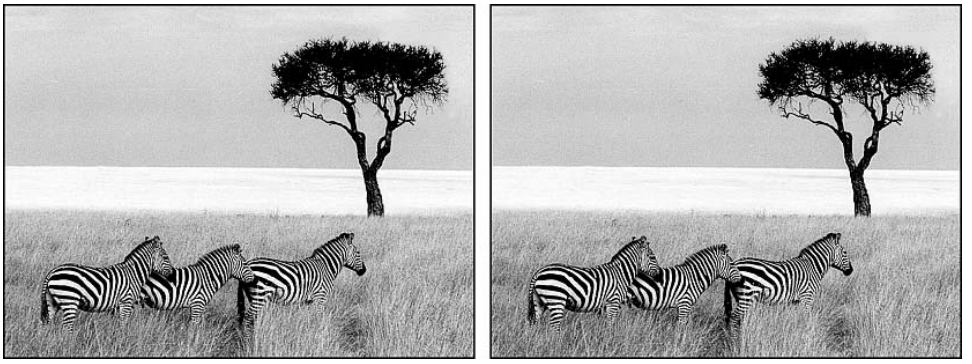
Steganography

In countries where censorship abounds, dissidents often try to use technology to evade it. Cryptography allows secret messages to be sent (although possibly not lawfully), but if the government thinks that Alice is a Bad Person, the mere fact that she is communicating with Bob may get him put in this category, too, as repressive governments understand the concept of transitive closure, even if they are short on mathematicians. Anonymous remailers can help, but if they are banned domestically and messages to foreign ones require a government export license, they cannot help much. But the Web can.

People who want to communicate secretly often try to hide the fact that any communication at all is taking place. The science of hiding messages is called **steganography**, from the Greek words for "covered writing." In fact, the ancient Greeks used it themselves. Herodotus wrote of a general who shaved the head of a messenger, tattooed a message on his scalp, and let the hair grow back before sending him off. Modern techniques are conceptually the same, only they have a higher bandwidth, lower latency, and do not require the services of a barber.

As a case in point, consider Fig. 8-54(a). This photograph, taken by one of the authors (AST) in Kenya, contains three zebras contemplating an acacia tree. Fig. 8-54(b) appears to be the same three zebras and acacia tree, but it has an extra added attraction. It contains the complete, unabridged text of five of

Shakespeare's plays embedded in it: *Hamlet*, *King Lear*, *Macbeth*, *The Merchant of Venice*, and *Julius Caesar*. Together, these plays total over 700 KB of text.



(a) (b)

Figure 8-54. (a) Three zebras and a tree. (b) Three zebras, a tree, and the complete text of five plays by William Shakespeare.

How does this steganographic channel work? The original color image is 1024×768 pixels. Each pixel consists of three 8-bit numbers, one each for the red, green, and blue intensity of that pixel. The pixel's color is formed by the linear superposition of the three colors. The steganographic encoding method uses the low-order bit of each RGB color value as a covert channel. Thus, each pixel has room for 3 bits of secret information, 1 in the red value, 1 in the green value, and 1 in the blue value. With an image of this size, up to $1024 \times 768 \times 3$ bits or 294,912 bytes of secret information can be stored in it.

The full text of the five plays and a short notice add up to 734,891 bytes. This text was first compressed to about 274 KB using a standard compression algorithm. The compressed output was then encrypted using IDEA and inserted into the low-order bits of each color value. As can be seen (or actually, cannot be seen), the existence of the information is completely invisible. It is equally invisible in the large, full-color version of the photo. The eye cannot easily distinguish 21-bit color from 24-bit color.

Viewing the two images in black and white with low resolution does not do justice to how powerful the technique is. To get a better feel for how steganography works, we have prepared a demonstration, including the full-color high-resolution image of Fig. 8-54(b) with the five plays embedded in it. The demonstration, including tools for inserting and extracting text into images, can be found at the book's Web site.

To use steganography for undetected communication, dissidents could create a Web site bursting with politically correct pictures, such as photographs of the Great Leader, local sports, movie, and television stars, etc. Of course, the pictures would be riddled with steganographic messages. If the messages were first

compressed and then encrypted, even someone who suspected their presence would have immense difficulty in distinguishing the messages from white noise. Of course, the images should be fresh scans; copying a picture from the Internet and changing some of the bits is a dead giveaway.

Images are by no means the only carrier for steganographic messages. Audio files also work fine. Hidden information can be carried in a voice-over-IP call by manipulating the packet delays, distorting the audio, or even in the header fields of packets (Lubacz et al., 2010). Even the layout and ordering of tags in an HTML file can carry information.

Although we have examined steganography in the context of free speech, it has numerous other uses. One common use is for the owners of images to encode secret messages in them stating their ownership rights. If such an image is stolen and placed on a Web site, the lawful owner can reveal the steganographic message in court to prove whose image it is. This technique is called **watermarking**. It is discussed in Piva et al. (2002).

For more on steganography, see Wayner (2008).

8.10.3 Copyright

Privacy and censorship are just two areas where technology meets public policy. A third one is the copyright law. **Copyright** is granting to the creators of **IP** (**Intellectual Property**), including writers, poets, artists, composers, musicians, photographers, cinematographers, choreographers, and others, the exclusive right to exploit their IP for some period of time, typically the life of the author plus 50 years or 75 years in the case of corporate ownership. After the copyright of a work expires, it passes into the public domain and anyone can use or sell it as they wish. The Gutenberg Project (*www.promo.net/pg*), for example, has placed thousands of public-domain works (e.g., by Shakespeare, Twain, and Dickens) on the Web. In 1998, the U.S. Congress extended copyright in the U.S. by another 20 years at the request of Hollywood, which claimed that without an extension nobody would create anything any more. By way of contrast, patents last for only 20 years and people still invent things.

Copyright came to the forefront when Napster, a music-swapping service, had 50 million members. Although Napster did not actually copy any music, the courts held that its holding a central database of who had which song was contributory infringement, that is, it was helping other people infringe. While nobody seriously claims copyright is a bad idea (although many claim that the term is far too long, favoring big corporations over the public), the next generation of music sharing is already raising major ethical issues.

For example, consider a peer-to-peer network in which people share legal files (public-domain music, home videos, religious tracts that are not trade secrets, etc.) and perhaps a few that are copyrighted. Assume that everyone is online all the time via ADSL or cable. Each machine has an index of what is on the hard

disk, plus a list of other members. Someone looking for a specific item can pick a random member and see if he has it. If not, he can check out all the members in that person's list, and all the members in their lists, and so on. Computers are very good at this kind of work. Having found the item, the requester just copies it.

If the work is copyrighted, chances are the requester is infringing (although for international transfers, the question of whose law applies matters because in some countries uploading is illegal but downloading is not). But what about the supplier? Is it a crime to keep music you have paid for and legally downloaded on your hard disk where others might find it? If you have an unlocked cabin in the country and an IP thief sneaks in carrying a notebook computer and scanner, scans a copyrighted book to the notebook's hard disk, and sneaks out, are *you* guilty of the crime of failing to protect someone else's copyright?

But there is more trouble brewing on the copyright front. There is a huge battle going on now between Hollywood and the computer industry. The former wants stringent protection of all intellectual property but the latter does not want to be Hollywood's policeman. In October 1998, Congress passed the **DMCA** (**Digital Millennium Copyright Act**), which makes it a crime to circumvent any protection mechanism present in a copyrighted work or to tell others how to circumvent it. Similar legislation has been enacted in the European Union. While virtually no one thinks that pirates in the Far East should be allowed to duplicate copyrighted works, many people think that the DMCA completely shifts the balance between the copyright owner's interest and the public interest.

A case in point: in September 2000, a music industry consortium charged with building an unbreakable system for selling music online sponsored a contest inviting people to try to break the system (which is precisely the right thing to do with any new security system). A team of security researchers from several universities, led by Prof. Edward Felten of Princeton, took up the challenge and broke the system. They then wrote a paper about their findings and submitted it to a USENIX security conference, where it underwent peer review and was accepted. Before the paper was to be presented, Felten received a letter from the Recording Industry Association of America that threatened to sue the authors under the DMCA if they published the paper.

Their response was to file a lawsuit asking a federal court to rule on whether publishing scientific papers on security research was still legal. Fearing a definitive court ruling against it, the industry withdrew its threat and the court dismissed Felten's suit. No doubt the industry was motivated by the weakness of its case: it had invited people to try to break its system and then threatened to sue some of them for accepting its own challenge. With the threat withdrawn, the paper was published (Craver et al., 2001). A new confrontation is virtually certain.

Meanwhile, pirated music and movies have fueled the massive growth of peer-to-peer networks. This has not pleased the copyright holders, who have used the DMCA to take action. There are now automated systems that search peer-to-peer networks and then fire off warnings to network operators and users who are

suspected of infringing copyright. In the United States, these warnings are known as **DMCA takedown notices**. This search is an arms' race because it is hard to reliably catch copyright infringers. Even your printer might be mistaken for a culprit (Piatek et al., 2008).

A related issue is the extent of the **fair use doctrine**, which has been established by court rulings in various countries. This doctrine says that purchasers of a copyrighted work have certain limited rights to copy the work, including the right to quote parts of it for scientific purposes, use it as teaching material in schools or colleges, and in some cases make backup copies for personal use in case the original medium fails. The tests for what constitutes fair use include (1) whether the use is commercial, (2) what percentage of the whole is being copied, and (3) the effect of the copying on sales of the work. Since the DMCA and similar laws within the European Union prohibit circumvention of copy protection schemes, these laws also prohibit legal fair use. In effect, the DMCA takes away historical rights from users to give content sellers more power. A major showdown is inevitable.

Another development in the works that dwarfs even the DMCA in its shifting of the balance between copyright owners and users is **trusted computing** as advocated by industry bodies such as the **TCG** (**Trusted Computing Group**), led by companies like Intel and Microsoft. The idea is to provide support for carefully monitoring user behavior in various ways (e.g., playing pirated music) at a level below the operating system in order to prohibit unwanted behavior. This is accomplished with a small chip, called a **TPM** (**Trusted Platform Module**), which it is difficult to tamper with. Most PCs sold nowadays come equipped with a TPM. The system allows software written by content owners to manipulate PCs in ways that users cannot change. This raises the question of who is trusted in trusted computing. Certainly, it is not the user. Needless to say, the social consequences of this scheme are immense. It is nice that the industry is finally paying attention to security, but it is lamentable that the driver is enforcing copyright law rather than dealing with viruses, crackers, intruders, and other security issues that most people are concerned about.

In short, the lawmakers and lawyers will be busy balancing the economic interests of copyright owners with the public interest for years to come. Cyberspace is no different from meatspace: it constantly pits one group against another, resulting in power struggles, litigation, and (hopefully) eventually some kind of resolution, at least until some new disruptive technology comes along.

8.11 SUMMARY

Cryptography is a tool that can be used to keep information confidential and to ensure its integrity and authenticity. All modern cryptographic systems are based on Kerckhoff's principle of having a publicly known algorithm and a secret

key. Many cryptographic algorithms use complex transformations involving substitutions and permutations to transform the plaintext into the ciphertext. However, if quantum cryptography can be made practical, the use of one-time pads may provide truly unbreakable cryptosystems.

Cryptographic algorithms can be divided into symmetric-key algorithms and public-key algorithms. Symmetric-key algorithms mangle the bits in a series of rounds parameterized by the key to turn the plaintext into the ciphertext. AES (Rijndael) and triple DES are the most popular symmetric-key algorithms at present. These algorithms can be used in electronic code book mode, cipher block chaining mode, stream cipher mode, counter mode, and others.

Public-key algorithms have the property that different keys are used for encryption and decryption and that the decryption key cannot be derived from the encryption key. These properties make it possible to publish the public key. The main public-key algorithm is RSA, which derives its strength from the fact that it is very difficult to factor large numbers.

Legal, commercial, and other documents need to be signed. Accordingly, various schemes have been devised for digital signatures, using both symmetric-key and public-key algorithms. Commonly, messages to be signed are hashed using algorithms such as SHA-1, and then the hashes are signed rather than the original messages.

Public-key management can be done using certificates, which are documents that bind a principal to a public key. Certificates are signed by a trusted authority or by someone (recursively) approved by a trusted authority. The root of the chain has to be obtained in advance, but browsers generally have many root certificates built into them.

These cryptographic tools can be used to secure network traffic. IPsec operates in the network layer, encrypting packet flows from host to host. Firewalls can screen traffic going into or out of an organization, often based on the protocol and port used. Virtual private networks can simulate an old leased-line network to provide certain desirable security properties. Finally, wireless networks need good security lest everyone read all the messages, and protocols like 802.11i provide it.

When two parties establish a session, they have to authenticate each other and, if need be, establish a shared session key. Various authentication protocols exist, including some that use a trusted third party, Diffie-Hellman, Kerberos, and public-key cryptography.

Email security can be achieved by a combination of the techniques we have studied in this chapter. PGP, for example, compresses messages, then encrypts them with a secret key and sends the secret key encrypted with the receiver's public key. In addition, it also hashes the message and sends the signed hash to verify message integrity.

Web security is also an important topic, starting with secure naming. DNSsec provides a way to prevent DNS spoofing. Most e-commerce Web sites use

SSL/TLS to establish secure, authenticated sessions between the client and server. Various techniques are used to deal with mobile code, especially sandboxing and code signing.

The Internet raises many issues in which technology interacts strongly with public policy. Some of the areas include privacy, freedom of speech, and copyright.

PROBLEMS

1. Break the following monoalphabetic substitution cipher. The plaintext, consisting of letters only, is an excerpt from a poem by Lewis Carroll.

mvyy bek mnyx n yvjjyr snijrh invq n muvjvdt je n idnvy
jurhri n fehfevir pyeir oruvdq ki ndq uri jhrnqvdt ed zb jnvy
Irr uem rntrhyb jur yeoijrhi ndq jur jkhjyri nyy nqlndpr
Jurb nhr mnvjvdt ed jur iuvdtyr mvyy bek pezr ndq wevd jur qndpr
mvyy bek, medj bek, mvyy bek, medj bek, mvyy bek wevd jur qndpr
mvyy bek, medj bek, mvyy bek, medj bek, medj bek wevd jur qndpr

2. An affine cipher is a version of a monoalphabetic substitution cipher, in which the letters of an alphabet of size m are first map to the integers in the range 0 to m-1. Subsequently, the integer representing each plaintext letter is transformed to an integer representing the corresponding cipher text letter. The encryption function for a single letter is $E(x) = (ax + b) \bmod m$, where m is the size of the alphabet and a and b are the key of the cipher, and are co-prime. Trudy finds out that Bob generated a ciphertext using an affine cipher. She gets a copy of the ciphertext, and finds out that the most frequent letter of the ciphertext is 'R', and the second most frequent letter of the ciphertext is 'K'. Show how Trudy can break the code and retrieve the plaintext.

3. Break the following columnar transposition cipher. The plaintext is taken from a popular computer textbook, so "computer" is a probable word. The plaintext consists entirely of letters (no spaces). The ciphertext is broken up into blocks of five characters for readability.

aauan cvlre rurnn dltme aeepb ytust iceat npmey iicgo gorch srsoc
nntii imiha oofpa gsivt tpsit lbolr otoex

4. Alice used a transposition cipher to encrypt her messages to Bob. For added security, she encrypted the transposition cipher key using a substitution cipher, and kept the encrypted cipher in her computer. Trudy managed to get hold of the encrypted transposition cipher key. Can Trudy decipher Alice's messages to Bob? Why or why not?

5. Find a 77-bit one-time pad that generates the text "Hello World" from the ciphertext of Fig. 8-4.

6. You are a spy, and, conveniently, have a library with an infinite number of books at your disposal. Your operator also has such a library at his disposal. You have agreed

to use *Lord of the Rings* as a one-time pad. Explain how you could use these assets to generate an infinitely long one-time pad.

7. Quantum cryptography requires having a photon gun that can, on demand, fire a single photon carrying 1 bit. In this problem, calculate how many photons a bit carries on a 250-Gbps fiber link. Assume that the length of a photon is equal to its wavelength, which for purposes of this problem, is 1 micron. The speed of light in fiber is 20 cm/nsec.

8. If Trudy captures and regenerates photons when quantum cryptography is in use, she will get some of them wrong and cause errors to appear in Bob's one-time pad. What fraction of Bob's one-time pad bits will be in error, on average?

9. A fundamental cryptographic principle states that all messages must have redundancy. But we also know that redundancy helps an intruder tell if a guessed key is correct. Consider two forms of redundancy. First, the initial n bits of the plaintext contain a known pattern. Second, the final n bits of the message contain a hash over the message. From a security point of view, are these two equivalent? Discuss your answer.

10. In Fig. 8-6, the P-boxes and S-boxes alternate. Although this arrangement is esthetically pleasing, is it any more secure than first having all the P-boxes and then all the S-boxes? Discuss your answer.

11. Design an attack on DES based on the knowledge that the plaintext consists exclusively of uppercase ASCII letters, plus space, comma, period, semicolon, carriage return, and line feed. Nothing is known about the plaintext parity bits.

12. In the text, we computed that a cipher-breaking machine with a million processors that could analyze a key in 1 nanosecond would take 10^{16} years to break the 128-bit version of AES. Let us compute how long it will take for this time to get down to 1 year, still along time, of course. To achieve this goal, we need computers to be 10^{16} times faster. If Moore's Law (computing power doubles every 18 months) continues to hold, how many years will it take before a parallel computer can get the cipher-breaking time down to a year?

13. AES supports a 256-bit key. How many keys does AES-256 have? See if you can find some number in physics, chemistry, or astronomy of about the same size. Use the Internet to help search for big numbers. Draw a conclusion from your research.

14. Suppose that a message has been encrypted using DES in counter mode. One bit of ciphertext in block C_i is accidentally transformed from a 0 to a 1 during transmission. How much plaintext will be garbled as a result?

15. Now consider ciphertext block chaining again. Instead of a single 0 bit being transformed into a 1 bit, an extra 0 bit is inserted into the ciphertext stream after block C_i. How much plaintext will be garbled as a result?

16. Compare cipher block chaining with cipher feedback mode in terms of the number of encryption operations needed to transmit a large file. Which one is more efficient and by how much?

17. Using the RSA public key cryptosystem, with $a = 1$, $b = 2 \cdots$ $y = 25$, $z = 26$.
(a) If $p = 5$ and $q = 13$, list five legal values for d.

(b) If $p = 5$, $q = 31$, and $d = 37$, find e.
(c) Using $p = 3$, $q = 11$, and $d = 9$, find e and encrypt "hello".

18. Alice and Bob use RSA public key encryption in order to communicate between them. Trudy finds out that Alice and Bob shared one of the primes used to determine the number n of their public key pairs. In other words, Trudy found out that $n_a = p_a \times q$ and $n_b = p_b \times q$. How can Trudy use this information to break Alice's code?

19. Consider the use of counter mode, as shown in Fig. 8-15, but with $IV = 0$. Does the use of 0 threaten the security of the cipher in general?

20. In Fig. 8-20, we see how Alice can send Bob a signed message. If Trudy replaces P, Bob can detect it. But what happens if Trudy replaces both P and the signature?

21. Digital signatures have a potential weakness due to lazy users. In e-commerce transactions, a contract might be drawn up and the user asked to sign its SHA-1 hash. If the user does not actually verify that the contract and hash correspond, the user may inadvertently sign a different contract. Suppose that the Mafia try to exploit this weakness to make some money. They set up a pay Web site (e.g., pornography, gambling, etc.) and ask new customers for a credit card number. Then they send over a contract saying that the customer wishes to use their service and pay by credit card and ask the customer to sign it, knowing that most of them will just sign without verifying that the contract and hash agree. Show how the Mafia can buy diamonds from a legitimate Internet jeweler and charge them to unsuspecting customers.

22. A math class has 25 students. Assuming that all of the students were born in the first half of the year—between January 1st and June 30th— what is the probability that at least two students have the same birthday? Assume that nobody was born on leap day, so there are 181 possible birthdays.

23. After Ellen confessed to Marilyn about tricking her in the matter of Tom's tenure, Marilyn resolved to avoid this problem by dictating the contents of future messages into a dictating machine and having her new secretary just type them in. Marilyn then planned to examine the messages on her terminal after they had been typed in to make sure they contained her exact words. Can the new secretary still use the birthday attack to falsify a message, and if so, how? *Hint*: She can.

24. Consider the failed attempt of Alice to get Bob's public key in Fig. 8-23. Suppose that Bob and Alice already share a secret key, but Alice still wants Bob's public key. Is there now a way to get it securely? If so, how?

25. Alice wants to communicate with Bob, using public-key cryptography. She establishes a connection to someone she hopes is Bob. She asks him for his public key and he sends it to her in plaintext along with an X.509 certificate signed by the root CA. Alice already has the public key of the root CA. What steps does Alice carry out to verify that she is talking to Bob? Assume that Bob does not care who he is talking to (e.g., Bob is some kind of public service).

26. Suppose that a system uses PKI based on a tree-structured hierarchy of CAs. Alice wants to communicate with Bob, and receives a certificate from Bob signed by a CA X after establishing a communication channel with Bob. Suppose Alice has never heard of X. What steps does Alice take to verify that she is talking to Bob?

27. Can IPsec using AH be used in transport mode if one of the machines is behind a NAT box? Explain your answer.

28. Alice wants to send a message to Bob using SHA-1 hashes. She consults with you regarding the appropriate signature algorithm to be used. What would you suggest?

29. Give one reason why a firewall might be configured to inspect incoming traffic. Give one reason why it might be configured to inspect outgoing traffic. Do you think the inspections are likely to be successful?

30. Suppose an organization uses VPN to securely connect its sites over the Internet. Jim, a user in the organization, uses the VPN to communicate with his boss, Mary. Describe one type of communication between Jim and Mary which would not require use of encryption or other security mechanism, and another type of communication which would require encryption or other security mechanisms. Explain your answer.

31. Change one message in the protocol of Fig. 8-34 in a minor way to make it resistant to the reflection attack. Explain why your change works.

32. The Diffie-Hellman key exchange is being used to establish a secret key between Alice and Bob. Alice sends Bob (227, 5, 82). Bob responds with (125). Alice's secret number, x, is 12, and Bob's secret number, y, is 3. Show how Alice and Bob compute the secret key.

33. Two users can establish a shared secret key using the Diffie-Hellman algorithm, even if they have never met, share no secrets, and have no certificates
(a) Explain how this algorithm is susceptible to a man-in-the-middle attack.
(b) How would this susceptibility change if n or g were secret?

34. In the protocol of Fig. 8-39, why is A sent in plaintext along with the encrypted session key?

35. In the Needham-Schroeder protocol, Alice generates two challenges, R_A and R_{A2}. This seems like overkill. Would one not have done the job?

36. Suppose an organization uses Kerberos for authentication. In terms of security and service availability, what is the effect if AS or TGS goes down?

37. Alice is using the public-key authentication protocol of Fig. 8-43 to authenticate communication with Bob. However, when sending message 7, Alice forgot to encrypt R_B. Trudy now knows the value of R_B. Do Alice and Bob need to repeat the authentication procedure with new parameters in order to ensure secure communication? Explain your answer.

38. In the public-key authentication protocol of Fig. 8-43, in message 7, R_B is encrypted with K_S. Is this encryption necessary, or would it have been adequate to send it back in plaintext? Explain your answer.

39. Point-of-sale terminals that use magnetic-stripe cards and PIN codes have a fatal flaw: a malicious merchant can modify his card reader to log all the information on the card and the PIN code in order to post additional (fake) transactions in the future. Next generation terminals will use cards with a complete CPU, keyboard, and tiny display on the card. Devise a protocol for this system that malicious merchants cannot break.

40. Is it possible to multicast a PGP message? What restrictions would apply?

41. Assuming that everyone on the Internet used PGP, could a PGP message be sent to an arbitrary Internet address and be decoded correctly by all concerned? Discuss your answer.

42. The attack shown in Fig. 8-47 leaves out one step. The step is not needed for the spoof to work, but including it might reduce potential suspicion after the fact. What is the missing step?

43. The SSL data transport protocol involves two nonces as well as a premaster key. What value, if any, does using the nonces have?

44. Consider an image of 2048×512 pixels. You want to encrypt a file sized 2.5 MB. What fraction of the file can you encrypt in this image? What fraction would you be able to encrypt if you compressed the file to a quarter of its original size? Show your calculations.

45. The image of Fig. 8-54(b) contains the ASCII text of five plays by Shakespeare. Would it be possible to hide music among the zebras instead of text? If so, how would it work and how much could you hide in this picture? If not, why not?

46. You are given a text file of size 60 MB, which is to be encrypted using steganography in the low-order bits of each color in an image file. What size image would be required in order to encrypt the entire file? What size would be needed if the file were first compressed to a third of its original size? Give your answer in pixels, and show your calculations. Assume that the images have an aspect ratio of 3:2, for example, 3000×2000 pixels.

47. Alice was a heavy user of a type 1 anonymous remailer. She would post many messages to her favorite newsgroup, *alt.fanclub.alice*, and everyone would know they all came from Alice because they all bore the same pseudonym. Assuming that the remailer worked correctly, Trudy could not impersonate Alice. After type 1 remailers were all shut down, Alice switched to a cypherpunk remailer and started a new thread in her newsgroup. Devise a way for her to prevent Trudy from posting new messages to the newsgroup, impersonating Alice.

48. Search the Internet for an interesting case involving privacy and write a one-page report on it.

49. Search the Internet for some court case involving copyright versus fair use and write a 1-page report summarizing your findings.

50. Write a program that encrypts its input by XORing it with a keystream. Find or write as good a random number generator as you can to generate the keystream. The program should act as a filter, taking plaintext on standard input and producing ciphertext on standard output (and vice versa). The program should take one parameter, the key that seeds the random number generator.

51. Write a procedure that computes the SHA-1 hash of a block of data. The procedure should have two parameters: a pointer to the input buffer and a pointer to a 20-byte output buffer. To see the exact specification of SHA-1, search the Internet for FIPS 180-1, which is the full specification.

52. Write a function that accepts a stream of ASCII characters and encrypts this input using a substitution cipher with the Cipher Block Chaining mode. The block size should be 8 bytes. The program should take plaintext from the standard input and print the ciphertext on the standard output. For this problem, you are allowed to select any reasonable system to determine that the end of the input is reached, and/or when padding should be applied to complete the block. You may select any output format, as long as it is unambiguous. The program should receive two parameters:

1. A pointer to the initializing vector; and
2. A number, k, representing the substitution cipher shift, such that each ASCII character would be encrypted by the kth character ahead of it in the alphabet.

For example, if $x = 3$, then A is encoded by D, B is encoded by E etc. Make reasonable assumptions with respect to reaching the last character in the ASCII set. Make sure to document clearly in your code any assumptions you make about the input and encryption algorithm.

53. The purpose of this problem is to give you a better understanding as to the mechanisms of RSA. Write a function that receives as its parameters primes p and q, calculates public and private RSA keys using these parameters, and outputs n, z, d and e as printouts to the standard output. The function should also accept a stream of ASCII characters and encrypt this input using the calculated RSA keys. The program should take plaintext from the standard input and print the ciphertext to the standard output. The encryption should be carried out character-wise, that is, take each character in the input and encrypt it independently of other characters in the input. For this problem, you are allowed to select any reasonable system to determine that the end of the input is reached. You may select any output format, as long as it is unambiguous. Make sure to document clearly in your code any assumptions you make about the input and encryption algorithm.

To save space as well as decrease boot time, the Linux kernel image is a compressed file that is extracted after it is loaded into memory. During the boot process, the boot loader typically creates a temporary RAM file system, known as `initramfs`. This file system contains necessary drivers and kernel modules that must be installed to support the ***real*** root file system (which is not in main memory). Once the kernel has started and the necessary drivers are installed, the kernel switches the root file system from the temporary RAM location to the appropriate root file system location. Finally, Linux creates the `systemd` process, the initial process in the system, and then starts other services (for example, a web server and/or database). Ultimately, the system will present the user with a login prompt. In Section 11.5.2, we describe the boot process for Windows.

It is worthwhile to note that the booting mechanism is not independent from the boot loader. Therefore, there are specific versions of the GRUB boot loader for BIOS and UEFI, and the firmware must know as well which specific bootloader is to be used.

The boot process for mobile systems is slightly different from that for traditional PCs. For example, although its kernel is Linux-based, Android does not use GRUB and instead leaves it up to vendors to provide boot loaders. The most common Android boot loader is LK (for "little kernel"). Android systems use the same compressed kernel image as Linux, as well as an initial RAM file system. However, whereas Linux discards the `initramfs` once all necessary drivers have been loaded, Android maintains `initramfs` as the root file system for the device. Once the kernel has been loaded and the root file system mounted, Android starts the `init` process and creates a number of services before displaying the home screen.

Finally, boot loaders for most operating systems—including Windows, Linux, and macOS, as well as both iOS and Android—provide booting into **recovery mode** or **single-user mode** for diagnosing hardware issues, fixing corrupt file systems, and even reinstalling the operating system. In addition to hardware failures, computer systems can suffer from software errors and poor operating-system performance, which we consider in the following section.

2.10 Operating-System Debugging

We have mentioned debugging from time to time in this chapter. Here, we take a closer look. Broadly, **debugging** is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include **performance tuning**, which seeks to improve performance by removing processing **bottlenecks**. In this section, we explore debugging process and kernel errors and performance problems. Hardware debugging is outside the scope of this text.

2.10.1 Failure Analysis

If a process fails, most operating systems write the error information to a **log fil** to alert system administrators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory of the process—and store it in a file for later analysis. (Memory was referred to as the

"core" in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process at the time of failure.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A failure in the kernel is called a **crash**. When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**.

Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks. Consider that a kernel failure in the file-system code would make it risky for the kernel to try to save its state to a file on the file system before rebooting. A common technique is to save the kernel's memory state to a section of disk set aside for this purpose that contains no file system. If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area. When the system reboots, a process runs to gather the data from that area and write it to a crash dump file within a file system for analysis. Obviously, such strategies would be unnecessary for debugging ordinary user-level processes.

2.10.2 Performance Monitoring and Tuning

We mentioned earlier that performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the operating system must have some means of computing and displaying measures of system behavior. Tools may be characterized as providing either ***per-process*** or ***system-wide*** observations. To make these observations, tools may use one of two approaches—***counters*** or ***tracing***. We explore each of these in the following sections.

2.10.2.1 Counters

Operating systems keep track of system activity through a series of counters, such as the number of system calls made or the number of operations performed to a network device or disk. The following are examples of Linux tools that use counters:

Per-Process

- `ps`—reports information for a single process or selection of processes
- `top`—reports real-time statistics for current processes

System-Wide

- `vmstat`—reports memory-usage statistics
- `netstat`—reports statistics for network interfaces
- `iostat`—reports I/O usage for disks

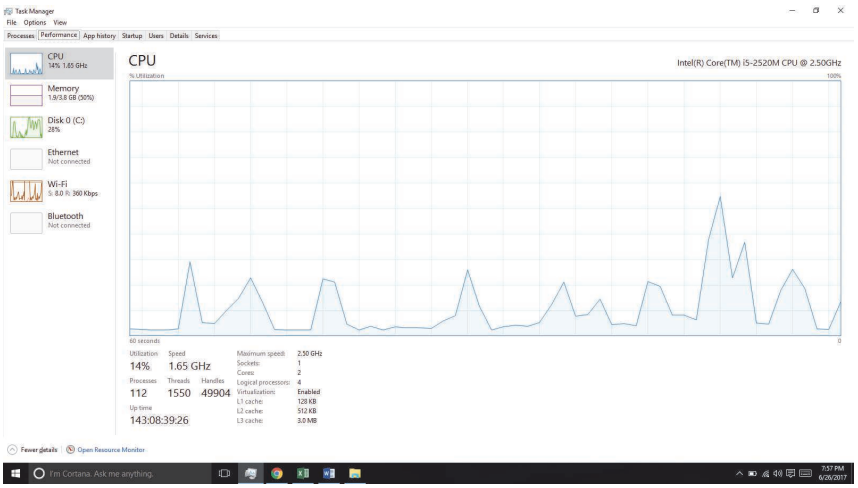


Figure 2.19 The Windows 10 task manager.

Most of the counter-based tools on Linux systems read statistics from the `/proc` file system. `/proc` is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various per-process as well as kernel statistics. The `/proc` file system is organized as a directory hierarchy, with the process (a unique integer value assigned to each process) appearing as a subdirectory below `/proc`. For example, the directory entry `/proc/2155` would contain per-process statistics for the process with an ID of 2155. There are `/proc` entries for various kernel statistics as well. In both this chapter and Chapter 3, we provide programming projects where you will create and access the `/proc` file system.

Windows systems provide the **Windows Task Manager**, a tool that includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager in Windows 10 appears in Figure 2.19.

2.10.3 Tracing

Whereas counter-based tools simply inquire on the current value of certain statistics that are maintained by the kernel, tracing tools collect data for a specific event—such as the steps involved in a system-call invocation.

The following are examples of Linux tools that trace events:

Per-Process

- `strace`—traces system calls invoked by a process
- `gdb`—a source-level debugger

System-Wide

- `perf`—a collection of Linux performance tools
- `tcpdump`—collects network packets

Kernighan's Law

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Making operating systems easier to understand, debug, and tune as they run is an active area of research and practice. A new generation of kernel-enabled performance analysis tools has made significant improvements in how this goal can be achieved. Next, we discuss BCC, a toolkit for dynamic kernel tracing in Linux.

2.10.4 BCC

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument their interactions. For that toolset to be truly useful, it must be able to debug any area of a system, including areas that were not written with debugging in mind, and do so without affecting system reliability. This toolset must also have a minimal performance impact—ideally it should have no impact when not in use and a proportional impact during use. The BCC toolkit meets these requirements and provides a dynamic, secure, low-impact debugging environment.

BCC (BPF Compiler Collection) is a rich toolkit that provides tracing features for Linux systems. BCC is a front-end interface to the eBPF (extended Berkeley Packet Filter) tool. The BPF technology was developed in the early 1990s for filtering traffic across a computer network. The "extended" BPF (eBPF) added various features to BPF. eBPF programs are written in a subset of C and are compiled into eBPF instructions, which can be dynamically inserted into a running Linux system. The eBPF instructions can be used to capture specific events (such as a certain system call being invoked) or to monitor system performance (such as the time required to perform disk I/O). To ensure that eBPF instructions are well behaved, they are passed through a **verifie** before being inserted into the running Linux kernel. The verifier checks to make sure that the instructions do not affect system performance or security.

Although eBPF provides a rich set of features for tracing within the Linux kernel, it traditionally has been very difficult to develop programs using its C interface. BCC was developed to make it easier to write tools using eBPF by providing a front-end interface in Python. A BCC tool is written in Python and it embeds C code that interfaces with the eBPF instrumentation, which in turn interfaces with the kernel. The BCC tool also compiles the C program into eBPF instructions and inserts it into the kernel using either probes or tracepoints, two techniques that allow tracing events in the Linux kernel.

The specifics of writing custom BCC tools are beyond the scope of this text, but the BCC package (which is installed on the Linux virtual machine we provide) provides a number of existing tools that monitor several areas

of activity in a running Linux kernel. As an example, the BCC `disksnoop` tool traces disk I/O activity. Entering the command

```
./disksnoop.py
```

generates the following example output:

```
TIME(s)          T    BYTES    LAT(ms)
1946.29186700    R    8            0.27
1946.33965000    R    8            0.26
1948.34585000    W    8192         0.96
1950.43251000    R    4096         0.56
1951.74121000    R    4096         0.35
```

This output tells us the timestamp when the I/O operation occurred, whether the I/O was a Read or Write operation, and how many bytes were involved in the I/O. The final column reflects the duration (expressed as latency or LAT) in milliseconds of the I/O.

Many of the tools provided by BCC can be used for specific applications, such as MySQL databases, as well as Java and Python programs. Probes can also be placed to monitor the activity of a specific process. For example, the command

```
./opensnoop -p 1225
```

will trace `open()` system calls performed only by the process with an identifier of 1225.

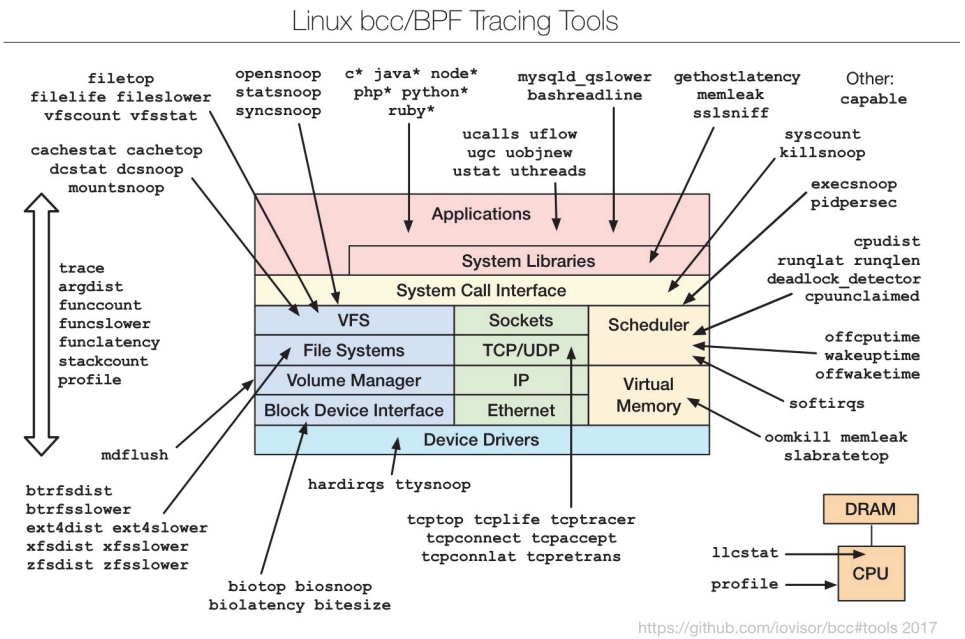


Figure 2.20 The BCC and eBPF tracing tools.

What makes BCC especially powerful is that its tools can be used on live production systems that are running critical applications without causing harm to the system. This is particularly useful for system administrators who must monitor system performance to identify possible bottlenecks or security exploits. Figure 2.20 illustrates the wide range of tools currently provided by BCC and eBPF and their ability to trace essentially any area of the Linux operating system. BCC is a rapidly changing technology with new features constantly being added.

2.11 Summary

- An operating system provides an environment for the execution of programs by providing services to users and programs.
- The three primary approaches for interacting with an operating system are (1) command interpreters, (2) graphical user interfaces, and (3) touch-screen interfaces.
- System calls provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The standard C library provides the system-call interface for UNIX and Linux systems.
- Operating systems also include a collection of system programs that provide utilities to users.
- A linker combines several relocatable object modules into a single binary executable file. A loader loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why applications are operating-system specific. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.
- A monolithic operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A layered operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. Although layered software systems have had some suc-