22 SECURITY

How do we keep intruders out of our computers? How do we keep them from impersonating us, or from listening in to our conversations or downloading our data? Computer security problems are in the news on almost a daily basis. In this chapter we take a look at just a few of the issues involved in building secure networks.

For our limited overview here, we will divide attacks into three categories:

1. Attacks that execute the intruder's code on the target computer
2. Attacks that extract data from the target, without code injection
3. Eavesdropping on or interfering with computer-to-computer communications

The first category is arguably the most serious; this usually amounts to a complete takeover, though occasionally the attacker's code is limited by operating-system privileges. A computer taken over this way is sometimes said to have been "owned". We discuss these attacks below in *22.1 Code-Execution Intrusion*. Perhaps the simplest form of such an attack is through stolen or guessed passwords to a system that offers remote login to command-shell accounts. More technical forms of attack may involve a virus, a buffer overflow (*22.2 Stack Buffer Overflow* and *22.3 Heap Buffer Overflow*), a protocol flaw (*22.1.2 Christmas Day Attack*), or some other software flaw (*22.1.1 The Morris Worm*).

In the second category are intrusions that do not involve remote code execution; a server application may be manipulated to give up data in ways that its designers did not foresee.

For example, in 2008 David Kernell gained access to the Yahoo email account of then-vice-presidential candidate Sarah Palin, by guessing or looking up the answers to the forgotten-password security questions for the account. One question was Palin's birthdate. Another was "where did you meet your spouse?", which, after some research and trial-and-error, Kernell was able to guess was "Wasilla High"; Palin grew up in and was at one point mayor of Wasilla, Alaska. Much has been made since of the idea that the answers to many security questions can be found on social-networking sites.

As a second example in this category, in 2010 Andrew "weev" Auernheimer and Daniel Spitler were charged in the "AT&T iPad hack". IPad owners who signed up for network service with AT&T had their iPad's ICC-ID recorded along with their other information. If one of these owners later revisited the AT&T website, the site would automatically request the iPad's ICC-ID and then populate the web form with the user's information. If a randomly selected ICC-ID were presented to the AT&T site that happened to match a real account, that user's name, phone number and email address would be returned. ICC-ID strings contain 20 decimal digits, but the individual-device portion of the identifier is much smaller and this brute-force attack yielded 114,000 accounts.

This attack is somewhat like a password intrusion, except that there was no support for running commands via the "compromised" accounts.

Auernheimer was convicted for this "intrusion" in November 2012, but his sentence was set aside on appeal in April 2014. Auernheimer's conviction remains controversial as the AT&T site never requested a password in the usual sense, though the site certainly released information not intended by its designers.

Finally, the third category here includes any form of eavesdropping. If the password for a login-shell account is obtained this way, a first-category attack may follow. The usual approach to prevention is the use of

encryption. Encryption is closely related to **secure authentication**; encryption and authentication are addressed below in *22.6 Secure Hashes* through *22.10 SSH and TLS*.

Encryption does not always work as desired. In 2006 intruders made off with 40 million credit-card records from **TJX Corp** by breaking the WEP Wi-Fi encryption (*22.7.7 Wi-Fi WEP Encryption Failure*) used by the company, and thus gaining access to account credentials and to file servers. Albert Gonzalez pleaded guilty to this intrusion in 2009. This was the largest retail credit-card breach until the Target hack of late 2013.

22.1 Code-Execution Intrusion

The most serious intrusions are usually those in which a vulnerability allows the attacker to run executable code on the target system.

The classic **computer virus** is broadly of this form, though usually without a network vulnerability: the user is tricked – often involving some form of social engineering – into running the attacker's program on the target computer; the program then makes itself at home more or less permanently. In one form of this attack, the user receives a file `interesting_picture.jpg.exe` or `IRS_deficiency_notice.pdf.exe`. The attack is made slightly easier by the default setting in Windows of not displaying the final file extension `.exe`.

Early viruses had to be completely self-contained, but, for networked targets, once an attacker is able to run some small initial executable then that program can in turn download additional malware. The target can also be further controlled via the network.

The reach of an executable-code intrusion may be limited by privileges on the target operating system; if I am operating a browser on my computer as user "pld" and an intruder takes advantage of a flaw in that browser, then the intruder's code will also run as "pld" and not as "root" or "Administrator". This may prevent the intruder from rewriting my kernel, though that is small comfort to me if my files are encrypted and held for ransom.

On servers, it is standard practice to run network services with the minimum privileges practical, though see *22.2.3 Defenses Against Buffer Overflows*.

Exactly what is "executable code" is surprisingly hard to state. Scripting languages usually qualify. In 2000, the ILOVEYOU virus began spreading on Windows systems; users received a file `LOVE-LETTER.TXT.vbs` (often with an enticing Subject: line such as "love letter for you"). The `.vbs` extension, again not displayed by default, meant that when the file was opened it was automatically run as a **v**isual **b**asic **s**cript. The ILOVEYOU virus was later attributed to Reonel Ramones and Onel de Guzman of the Philippines, though they were never prosecuted. The year before, the Melissa virus spread as an emailed Microsoft Word attachment; the executable component was a Word macro.

Under Windows, a number of configuration-file formats are effectively executable; among these are the program-information-file format .PIF and the screen-saver format .SCR.

22.1.1 The Morris Worm

The classic Morris Worm was launched on the infant Internet in 1988 by Robert Tappan Morris. Once one machine was infected, it would launch attacks against other machines, either on the same LAN or far away.

The worm used a number of techniques, including taking advantage of **implementation flaws** via stack buffer overflows (*22.2 Stack Buffer Overflow*). Two of the worm's techniques, however, had nothing to do with code injection. One worm module contained a dictionary of popular passwords that were used to try against various likely system accounts. Another module relied on a different kind of implementation vulnerability: a (broken) diagnostic feature of the `sendmail` email server. Someone could connect to the `sendmail` TCP port 25 and send the command `WIZ <password>`; that person would then get a shell and be able to execute arbitrary commands. It was the intent to require a legitimate `sendmail`-specific password, but an error in `sendmail`'s frozen-configuration-file processing meant that an empty password often worked.

22.1.2 Christmas Day Attack

The 1994 "Christmas day attack" (*12.10.1 ISNs and spoofing*) used a TCP **protocol weakness** combined with a common computer-trust arrangement to gain privileged login access to several computers at UCSD. Implementations can be fixed immediately, once the problem is understood, but protocol changes require considerable negotiation and review.

The so-called "rlogin" trust arrangement meant that computer A might be configured to trust requests for remote-command execution from computer B, often on the same subnet. But the ISN-spoofing attack meant that an attacker M could send a command request to A that would *appear* to come from the trusted peer B, at least until it was too late. The command might be as simple as "open up a shell connection to M". At some point the spoofed connection would fail, but by then the harmful command would have been executed. The only fix is to stop using rlogin. (Ironically, the ISN spoofing attack was discovered by Morris but was not used in the Morris worm above; see *[RTM85]*.)

Note that, as with the sendmail WIZ attack of the Morris worm, this attack did not involve network delivery of an executable fragment (a "shellcode").

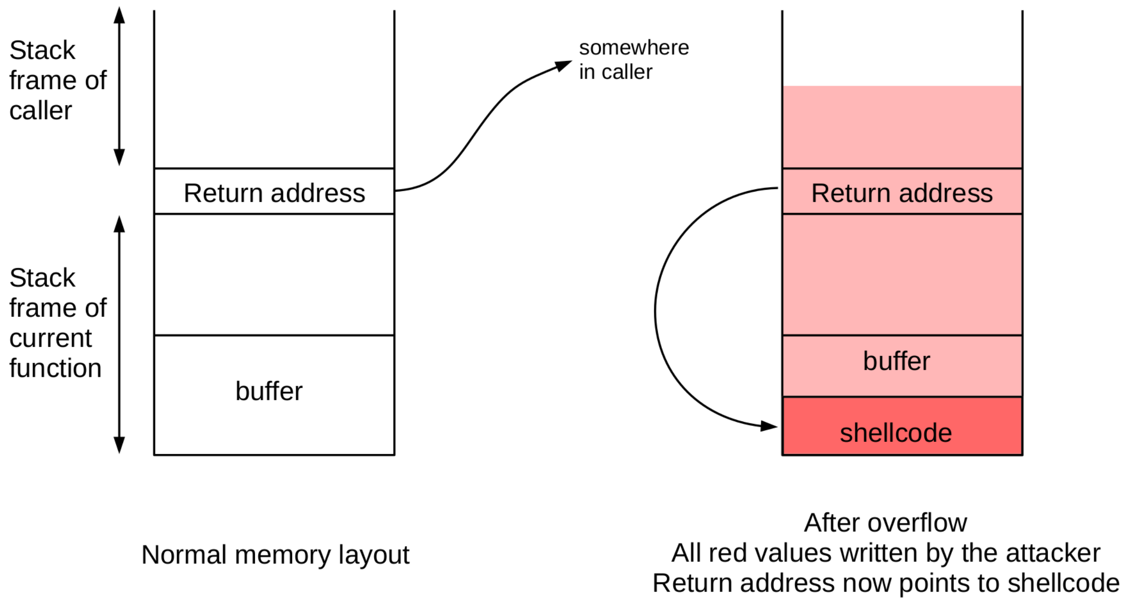
22.2 Stack Buffer Overflow

The stack buffer overflow is perhaps the classic way for an attacker to execute a short piece of machine code on a remote machine, thus compromising it. Such attacks are always due to an implementation flaw. A server application reads attacker-supplied data into a buffer, `buf`, of length `buflen`. Due to the flaw, however, the server reads more than `buflen` bytes of data, and the additional data is written into memory past the end of `buf`, corrupting memory. In the C language, there is no bounds checking with native arrays, and so such an overflow is not detected at the time it occurs. See *[AO96]* for early examples.

In most memory layouts, the stack grows downwards; that is, a function call creates a new stack frame with a numerically lower address. Array indexing, however, grows upwards: `buf[i+1]` is at a higher address than `buf[i]`. As a consequence, overwriting the buffer allows rewriting the most recent return address on the stack. A common goal for the attacker is to supply an overflowing buffer that does two things:

1. it includes a **shellcode** - a small snippet of machine code that, when executed, does something bad (traditionally but not necessarily by starting a shell with which the attacker can invoke arbitrary commands).
2. it overwrites the stack return address so that, when the current function exits, control is returned not to the caller but to the supplied shellcode.

In the diagram below, the left side shows the normal layout: the current stack frame has a buffer into which the attacker's message is read. When the current function exits, control is returned to the address stored in `return_address`.



The right side shows the result after the attacker has written shellcode to the buffer, and, by overflow, has also overwritten the `return_address` field on the stack so that it now points to the shellcode. When the function exits, control will be passed to the code at `return_address`, that is, to the shellcode rather than to the original caller. The shellcode is here shown below `return_address` in memory, but it can also be above it.

22.2.1 Return to libc

A variant attack is for the attacker to skip the shellcode, but instead to overwrite the stack return address with the address of a known library procedure. The libc library is popular here, making this known as the **return-to-libc** approach. The goal of the attacker is to identify a library procedure that, in the current context of the server process being attacked, will do something that the attacker finds useful.

One version of this attack is to overwrite the stack address with the address of the `system()` call, and to place on the stack just above this return address a pointer to the string "/bin/sh" (often present in the environment strings of the attacked process). When the current function exits, control branches to `system()`, which now thinks it has been called with parameter `/bin/sh`. A shell is launched.

Return-to-libc attacks often involve no shellcode at all. The attack of *22.3.2 A JPEG heap vulnerability* uses some return-to-libc elements but also does involve injected shellcode.

A practical problem with any form of the stack-overflow attack is knowing enough about the memory layout of the target machine so that the attacker can determine exactly where the shellcode is loaded. This is made simpler by standardized versions of Windows in which many library routines are at fixed, well-known addresses.

22.2.2 An Actual Stack-Overflow Example

Why the code here?

In many accounts of computer vulnerabilities, there is an understandable reluctance to explain the actual mechanics, lest some attacker learn how to exploit the flaw. This secrecy, however, sometimes has the unfortunate side-effect of making vulnerabilities seem almost magical. The goal in presenting this twenty-year-old example in detail is simply to strip away the aura of mystery surrounding many exploits.

To put together an actual example, we modify a version of TCP simplex-talk (*12.6 TCP simplex-talk*) written in the C language. On the server side, all we have to do is read the input with the infamous `gets(char * buf)`, which reads in data up to a newline (or NUL) character into the array `buf`, with no size restrictions. To be able to use `gets()` this way, we must arrange for the standard-input stream of the reading process to be the network connection. There is a version of `gets()` that reads from an arbitrary input stream, namely `fgets(char * buf, int bufsize, FILE *stream)`, but `fgets()` is not as vulnerable as `gets()` as it will not read more than `bufsize` characters into `buf`. (It is possible, of course, for a programmer to supply a value of `bufsize` much larger than the actual size of `buf[]`.)

One drawback of using `gets()` is that the shellcode string must not have any NULL (zero) bytes. Sometimes this takes some subterfuge; in the example below, the shellcode includes the string `"/bin/shNAAAABBBB"`; the `N` and the `BBBB` will be replaced with zeroes by the injected code itself, after it is read.

Alternatively, we can also have the server read its the data with the call

```
recv(int socket, char * buf, int bufsize, int flags)
```

but supply an *incorrect* (and much too large) value for the parameter `bufsize`. This approach has the practical advantage of allowing the attacker to supply a buffer with NUL characters (zero bytes) and with additional embedded newline characters.

On the client side – the attacker's side – we need to come up with a suitable shellcode and create a too-large buffer containing, in the correct place, the address of the start of the shellcode. Guessing this address used to be easy, in the days when every process always started with the same virtual-memory address. It is now much harder precisely to make this kind of buffer-overflow attack more difficult; we will cheat and have our server print out an address on startup that we can then supply to the client.

An attack like this depends on knowing the target operating system, the target cpu architecture (so as to provide an executable shellcode), the target memory layout, and something about the target server implementation (so the attacker knows what overflow to present, and when). Alas, all but the last of these are readily guessable. Once upon a time vulnerabilities in server implementations were discovered by reading source code, but it has long been possible to search for overflow opportunities making use only of the binary executable code.

The shellcode presented here assumes that the server system is running 32-bit code. 64-bit Linux systems can be configured to run 32-bit code as well, though a simpler alternative is often to create a 32-bit virtual machine to run the server side. It would also be possible to migrate the shellcode to 64-bit format.

22.2.2.1 The server

The overflow-vulnerable server, oserver.c, is more-or-less a C implementation of the tcp simplex-talk server of *12.6 TCP simplex-talk*. For the 32-bit shellcode below to work, the server must be run as a 32-bit program.

The server contains an explicit call to `bind()` which was handled implicitly by the `ServerSocket()` constructor in the Java version.

For each new connection, a new process to handle it is created with `fork()`; that new process then calls `process_connection()`. The `process_connection()` function then reads a line at a time from the client into a buffer `pcbuf` of 80 bytes. Unfortunately for the server, it may read well more than 80 bytes into `pcbuf`.

For the stack overflow to work, it is essential that the function that reads too much data into the buffer – thus corrupting the stack – must return. Therefore the protocol has been modified so that `process_connection()` returns if the arriving string begins with "quit".

We must also be careful that the stack overflow does not so badly corrupt any local variables that the code fails afterwards to continue running, even before returning. All local variables in `process_connection()` are overwritten by the overflow, so we save the socket itself in the global variable `gsock`.

We also call `setstdinout(gsock)` so that the standard input and standard output within `process_connection()` is the network connection. This allows the use of the notoriously vulnerable `gets()`, which reads from the standard input (alternatively, `recv()` or `fgets()` with an incorrect value for `bufsize` may be used). The call to `setstdinout()` also means that the shell launched by the shellcode will have its standard input and output correctly set up. We could, of course, make the appropriate `dup()`/`fcntl()` calls from the shellcode, but that would increase its complexity and size.

Because the server's standard output is now the TCP connection, it prints incoming strings to the terminal via the standard-error stream.

On startup, the server prints an address (that of `mbuf[]`) within its stack frame; we will refer to this as **mbuf_addr**. The attacking client must know this value. No real server is so accommodating as to print its internal addresses, but in the days before address randomization, *22.2.3.2 ASLR*, the server's stack address was typically easy to guess.

Whenever a connection is made, the server here also prints out the distance, in bytes, between the start of `mbuf[]` in `main()` and the start of `pcbuf` – the buffer that receives the overflow – in `process_connection()`. This latter number, which we will call **addr_diff**, is constant, and must be compiled into the exploit program (it does change if new variables are added to the server's `main()` or `process_connection()`). The actual address of `pcbuf[]` is thus mbuf_addr – addr_diff. This will be the address where the shellcode is located, and so is the address with which we want to overwrite the stack. We return to this below in *22.2.2.3 The exploit*, where we introduce a "NOPslide" so that the attacker does not have to get the address of `pcbuf[]` exactly right.

Linux provides some protection against overflow attacks (*22.2.3 Defenses Against Buffer Overflows*), so the server must disable these. As mentioned above, one protection, address randomization, we defeat by having the server print a stack address. The server must also be compiled with the `-fno-stack-protector` option to disable the stack canary of *22.2.3.1 Stack canary*, and the `-z execstack` option to disable making the stack (and other data areas) non-executable, *22.2.3.3 Making the stack non-executable*.

```
gcc -fno-stack-protector -z execstack -o oserver oserver.c
```

Even then we are dutifully warned by both the compiler and the linker:

```
warning: 'gets' is deprecated ....
warning: the 'gets' function is dangerous and should not be used.
```

In other words, getting this vulnerability still to work in 2014 takes a bit of effort.

The server here does work with the simplex-talk client of *12.6 TCP simplex-talk*, but with the #USE_GETS option enabled it does not handle a client exit gracefully, unless the client sends "quit".

22.2.2.2 The shellcode

The shellcode must be matched to the operating system and to the cpu architecture; we will assume Linux running on a 32-bit Intel x86. Our goal is to create a shellcode that launches /bin/sh. The approach here is taken from *[SH04]*.

The shellcode must be a string of machine instructions that is completely self-contained; data references cannot depend on the linker for address resolution. Not only must the code be position-independent, but the code together with its data must be position-independent.

So-called "Intel syntax" is used here, in which the destination operand comes first, *eg* mov eax,37. The nasm assembler is one of many that supports this format.

Direct system calls in Linux are made using interrupts, using the int 0x80 opcode and parameter. The x86 architecture has four general-purpose registers eax, ebx, ecx and edx; when invoking a system call via an interrupt, the first of these holds a code for the particular system routine to be invoked and the others hold up to three parameters. (Below, in *22.3.2 A JPEG heap vulnerability*, we will also make use of register edi.) The system call we want to make is

```
execve(char * filename, char *argv[], char *envp[])
```

We need eax to contain the numeric value 11, the 32-bit-Linux syscall value corresponding to execve (perhaps defined in /usr/include/i386-linux-gnu/asm/unistd_32.h). (64-bit Linux uses 59 as the syscall value for execve.) We load this with mov al 11; al is a shorthand for the low-order byte of register eax. We first zero eax by subtracting it from itself. We can also, of course, use mov eax 11, but then the 11 expands into four bytes 0x0b000000, and we want to avoid including NUL bytes in the code.

We also need ebx to point to the NUL-terminated string /bin/sh. The register ecx should point to an array of pointers ["/bin/sh", 0] in memory (the null-terminated argv[]), and edx should point to a null word in memory (the null-terminated envp[]). We include in the shellcode the string "/bin/shNAAAABBBB", and have the shellcode insert a NUL byte to replace the N and a zero word to replace the "BBBB", as the shellcode must contain no NUL bytes at the time it is read in by gets(). The shellcode will also replace the "AAAA" with the address of "/bin/sh". We then load ecx with the address of "AAAA" (now containing the address of "/bin/sh" followed by a zero word) and edx with the address of "BBBB" (now just a zero word).

Loading the address of a string is tricky in the x86 architecture. We want to calculate this address relative to the current instruction pointer IP, but no direct access is provided to IP. The workaround in the code below is to jump to `shellstring` near the end, but then invoke `call start`, where `start:` is the label for our main code up at the top. The action of `call start` is to push the address of the byte following `call start` onto the stack; this happens to be the address of `shellstring:`. Back up at `start:`, the `pop ebx` pops this address off the stack and leaves it in `ebx`, where we want it.

Our complete shellcode is as follows (the actual code is in shellcode.s):

```
jmp short shellstring
start:
pop ebx                  ;get the address of the string in ebx
sub eax, eax             ;zero eax by subtracting it from itself
mov [ebx+7 ], al         ;put a NUL byte where the N is in the string
mov [ebx+8 ], ebx        ;put the address of the string where the AAAA is
mov [ebx+12], eax        ;put a zero (NULL) word into where the BBBB is
mov al, 11               ;execve is syscall 11
lea ecx, [ebx+8]         ;load the address of where the AAAA was
lea edx, [ebx+12]        ;load the address of where the BBBB was, now NULL
int 0x80                 ;call the kernel. WE HAVE A SHELL!
shellstring:
call start               ;pushes address of string below and jumps to start:
db '/bin/shNAAAABBBB'    ;the string. AAAA and BBBB get filled in as above
```

We run this through the commands

```
nasm -f elf shellcode.s
ld -o shellcode shellcode.o
objdump -d shellcode
```

and then, creating a string with the bytes produced, come up with the following:

```
char shellcode[] =
       "\xeb\x16\x5b\x29\xc0\x88\x43\x07\x89\x5b\x08\x89"
       "\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80"
       "\xe8\xe5\xff\xff\xff/bin/sh/NAAAABBBB";
```

We can test this (on a 32-bit system) with a simple C program defining the above and including

```
int main(int argc, char **argv) {
    void (*func)() = (void (*)()) shellcode;
    func();
}
```

We can verify that the resulting shell has not inherited the parent environment with the `env` and `set` commands.

Additional shellcode examples can be found in *[AHLR07]*.

22.2.2.3 The exploit

Now we can assemble the actual attack. We start with a C implementation of the simplex-talk client, and add a feature so that when the input string is "doit", the client

- sends the oversized buffer containing the shellcode, terminated with a newline to make `gets()` happy
- also sends "quit", terminated with a newline, to force the server's `process_connection()` to return, and thus to transfer control to the code pointed to by the now-overwritten `return_address` field of the stack
- begins a loop – `copylines()` – to copy the local terminal's standard input to the network connection (hopefully now with a shell at the other end), and to copy the network connection to the local terminal's standard output

On startup, the client accepts a command-line parameter representing the address (in hex) of a variable close to the start of the server's `main()` stack frame. When the server starts, it prints this address out; we simply copy that when starting the client. A second command-line parameter is the server hostname.

The full client code is in netsploit.c.

All that remains is to describe the layout of the malicious oversized buffer, created by `buildbadbuf()`. We first compute our guess for the address of the start of the vulnerable buffer `pcbuf` in the server's `process_connection()` function: we take the address passed in on the command line, which is actually the address of `mbuf` in the server's `main()`, and add to it the known constant (`pcbuf` - `mbuf`). This latter value, 147 in the version tested by the author, is stored in netsploit.c's `BUF_OFFSET`.

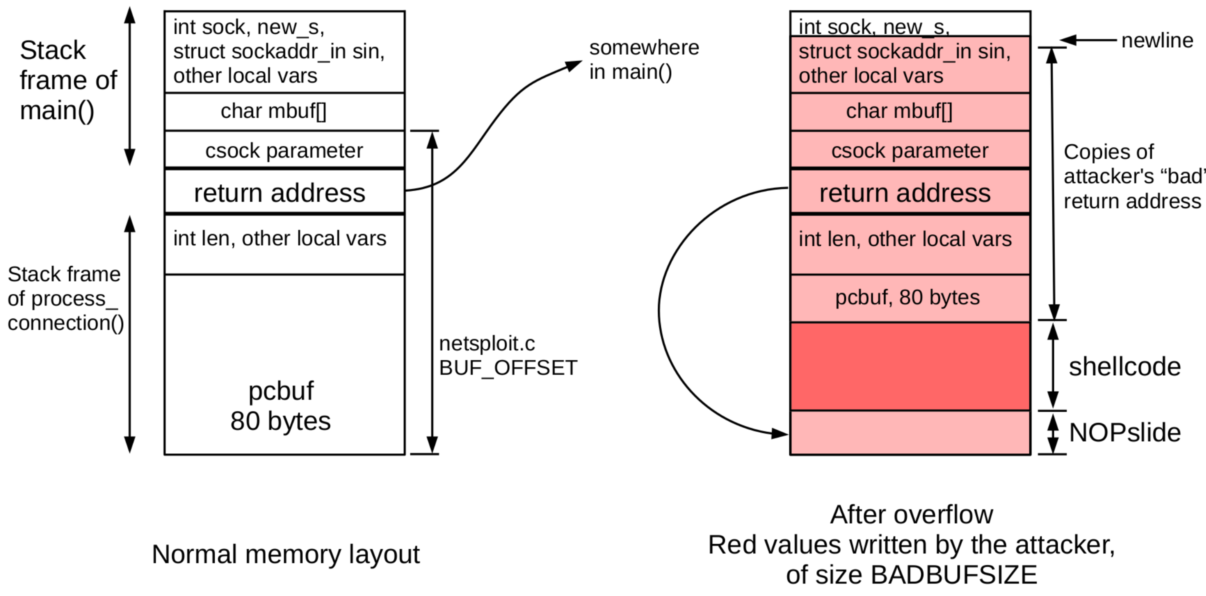
This calculation of the address of the server's `pcbuf` *should* be spot-on; if we now store our shellcode at the start of `pcbuf` and arrange for the server to jump to our calculated address, the shellcode should run. Real life, however, is not always so accommodating, so we introduce a **NOP slide**: we precede our shellcode with a run of NOP instructions. A jump anywhere into the NOPslide should lead right into the shellcode. In our example here, we make the NOPslide 20 bytes long, and add a fudge factor of between 0 and 20 to our calculated address (`FUDGE` is 10 in the actual code).

We need the attack buffer to be large enough that it overwrites the stack return-address field, but small enough that the server does not incur a segmentation fault when overfilling its buffer. We settle on a `BADBUFSIZE` of 161 (160 plus 1 byte for a final newline character); it should be comparable to but perhaps slightly larger than the `BUF_OFFSET` value of, in our case, 147).

The attack buffer is now

- 25 bytes of NOPs
- shellcode (46 bytes in the version above)
- 90 bytes worth of the repeated calculated address (`baseaddr-BUF_OFFSET+FUDGE` in the code
- 1 byte newline, as the server's `gets()` expects a newline-terminated string

Here is a diagram like the one above, but labeled for this particular attack. Not all memory regions are drawn to scale, and there are more addresses between the two stack frames than just `return address`.


Stack frame of main()
int sock, new_s, struct sockaddr_in sin, other local vars
char mbuf[]
csock parameter
return address
int len, other local vars
Stack frame of process_ connection()
pcbuf 80 bytes
netsploit.c BUF_OFFSET
somewhere in main()
Normal memory layout
newline
Copies of attacker's "bad" return address
pcbuf, 80 bytes
shellcode
NOPslide
After overflow
Red values written by the attacker, of size BADBUFSIZE

We double-check the bad buffer to make sure it contains no NUL bytes and no other newline characters.

If we wanted a longer NOPslide, we would have to hope there was room *above* the stack's return-address field. In that case, the attack buffer would consist of a bunch of repeated address guesses, then the NOPslide, and finally the shellcode.

After the command "doit", the netsploit client prompt changes to `1>`. We can then type `ls`, and, if the shellcode has successfully started, we get back a list of files on the server. As earlier, we can also type `env` and `set` to verify that the shell did not inherit its environment from any "normal" shell. Note that any shell commands that need to write to the stderr stream will fail, as this stream was not set up; this includes any mistyped commands.

22.2.3 Defenses Against Buffer Overflows

How to prevent this sort of attack? The most basic approach is to make sure that **array bounds** are never violated (and also that similar rules for the use of dynamically allocated memory, such as not using a block after it has been freed, are never violated). Some languages enforce this automatically through "memory-safe" semantics; while this is not a guarantee that programs are safe, it does eliminate an important class of vulnerabilities. In C, memory- and overflow-related bugs can be eliminated through careful programming, but the task is notoriously error-prone.

Another basic approach, applicable to almost all remote-code-execution attacks, is to make sure that the server runs with the minimum **permissions** possible. The server may not have write permission to anything of substance, and may in fact be run in a so-called "chroot" environment in which any access to the bulk of the server's filesystem is disabled.

One issue with this approach is that a server process on a unix-derived system that wants to listen on a port less than 1024 needs special privileges to open that port. Traditionally, the process would start with root privileges and, once the socket was opened, would downgrade its privileges with calls to `setgid()` and `setuid()`. This is safe in principle, but requires careful attention to the `man` pages; use of `seteuid()`, for example, allows the shellcode to recover the original privileges. Linux systems now support assigning to an unprivileged process any of several "capabilities" (see `man capabilities`). The one most relevant here is `CAP_NET_BIND_SERVICE`, which, if set, allows a process to open privileged ports. Still, to assign these capabilities still requires some privileged intervention when the process is started.

22.2.3.1 Stack canary

The underlying system can also provide some defenses. One of these, typically implemented within the compiler, is the **stack canary**: an additional word on the stack near the return address that is set to a pseudo-random value. A copy of this word is saved elsewhere. When the function calls `return` (either explicitly or implicitly), this word is checked to make sure the stack copy still agrees with the saved-elsewhere copy; a discrepancy indicates that the stack was overwritten.

In the `gcc` compiler, a stack canary can be enabled with the compiler option `-fstack-protector`. To compile the stack exploit in *22.2.2.3 The exploit*, we needed to add `-fno-stack-protector`.

22.2.3.2 ASLR

Another operating-system-based defense is **Address-Space Layout Randomization**, or ASLR. In its simplest form, this means that each time the server process is restarted, the stack will have a different starting address. For example, restarting the oserver program above five times yields addresses (in hex) of bf8d22b7, bf84ed87, bf977d87, bfcc5cb7 and bfa302d7. There are at least four hex digits (16 bits) of entropy here; if the server did not print its stack address it might take 2^{16} guesses for the attacker to succeed.

Still, 2^{16} guesses might take an attacker well under an hour. Worse, the attacker might simply create a stack-buffer-overflow attack with a very long NOPslide; the longer the NOPslide the more room for error in guessing the shellcode address. With a NOPslide of length 2^{10} = 1024 bits, guessing the correct stack address will take only 2^6 = 64 tries. (Some implementations squeeze out 19 bits of address-space entropy, meaning that guessing the correct address increases to 2^9 = 512 tries.)

For 64-bit systems, however, ASLR is much more effective. Brute-force guessing of the stack address takes a prohibitively long time.

ALSR also changes the heap layout and the location of libraries each time the server process is restarted. This is to prevent return-to-libc attacks, *22.2.1 Return to libc*. For a concrete example of an attacker's use of non-randomized library addresses, see *22.3.2 A JPEG heap vulnerability*.

On Linux systems, ASLR can be disabled by writing a 0 to /proc/sys/kernel/randomize_va_space; values 1 and 2 correspond to partial and full randomization.

Windows systems since Vista (2007) have had ASLR support, though earlier versions of the linker required the developer to request ASLR with the `/DYNAMICBASE` switch.

22.2.3.3 Making the stack non-executable

A more sophisticated idea, if the virtual-memory hardware supports it, is to mark those pages of memory allocated to the stack as *non-executable*, meaning that if the processor's instruction register is loaded with an address on those pages (due to branching to stack-based shellcode), a hardware-level exception will immediately occur. This immediately prevents attacks that place a shellcode on the stack, though return-to-libc attacks (*22.2.1 Return to libc*) are still possible.

In the x86 world, AMD introduced the per-page NX bit, for No eXecute, in their x86-64 architecture; CPUs with this architecture began appearing in 2003. Intel followed with its XD, for eXecute Disabled, bit. Essentially all x86-64 CPUs now provide hardware NX/XD support; support on 32-bit CPUs generally requires so-called Physical Address Extension mode.

The NX idea applies to all memory pages, not just stack pages. This sometimes causes problems with applications such as just-in-time compilation, where the code page is written and then immediately executed. As a result, it is common to support NX-bit disabling in software. On Linux systems, the compiler option `-z execstack` disables NX-bit protection; this was used above in *22.2.2.1 The server*. Windows has a similar `/NXCOMPAT` option for requesting NX-bit protection.

While a non-executable stack prevents the stack-overflow attack described above, injecting shellcode onto the heap is still potentially possible. The OpenBSD operating system introduced **write or execute** in 2003; this is abbreviated **W^X** after the use of "^" as the XOR operator in C. A memory page may be writable or executable, but not both. This is strong protection against virtually all shellcode-injection attacks, but may still be vulnerable to some return-to-libc attacks (*22.2.1 Return to libc*).

See [AHLR07], chapter 14, for some potential attack strategies against systems with non-executable pages.

22.3 Heap Buffer Overflow

As with stack overflows, heap overflows all rely on some software flaw that allows data to be written beyond the confines of the designated buffer. A buffer on the heap is subject to the same software-failure overflow prospects as a buffer on the stack. An important difference, however, is that buffers on the heap are not in clear proximity to an obvious return address. Despite that difference, heap overflows can also be used to enable remote-code-execution attacks.

Perhaps the simplest heap overflow is to take advantage of the fact that some heap pages contain executable code, representing application-loaded library pages. If the page with the overflowable buffer is pointed to by `p`, and the following page in memory pointed to by `q` contains code, then all an attacker has to do is to have the overflow fill the `q` page with a long NOPslide and a shellcode. When at some point a call is made to the code pointed to by `q`, the shellcode is executed instead. A drawback to this attack is that the layout of heap pages like this is seldom known. On the other hand, the fact that heap pages do sometimes legitimately contain executable code means that uniformly marking the heap as non-executable, as is commonly done with the stack, may not be an option.

22.3.1 A Linux heap vulnerability

We now describe an actual Linux heap vulnerability from 2003, following *[AB03]*, based on version 2.2.4 of the glibc library. The vulnerable server code is simply the following:

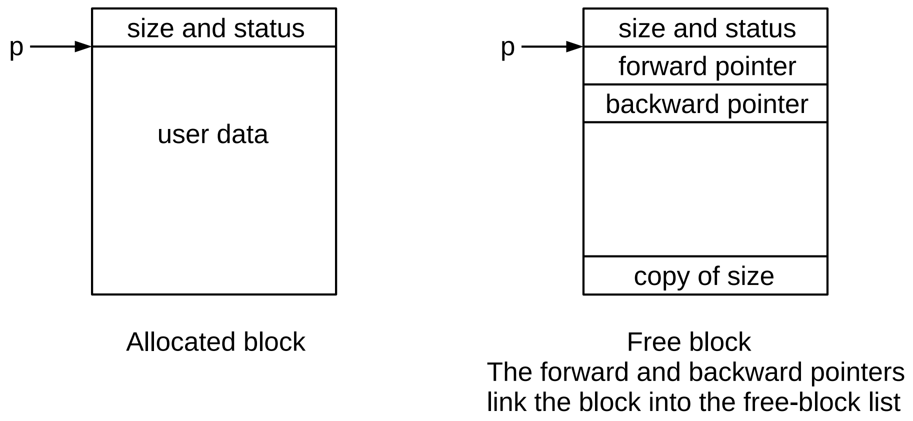
```
char *p = malloc(1024);
char *q = malloc(1024);
gets(p);                // read the attacker's input
free(q);                // block q is last allocated and first freed
free(p);
```

As with the stack-overflow example, the `gets(p)` results in an overflow from block p into block q, overwriting not just the data in block q but also the block headers maintained by `malloc()`. While there is no guarantee in general that block `q` will immediately follow block `p` in memory, in practice this usually happens unless there has been a great deal of previous allocate/free activity.

The vulnerability here relies on some features of the 2003-era (glibc-2.2.4) `malloc()`. All blocks are either allocated to the user or are free; free blocks are kept on a doubly linked list. We will assume the data portion of each block is always a power of 2; there is a separate free-block list for each block size. When two adjacent blocks are both free, `malloc()` coalesces them and moves the joined block to the free-block list of the next size up.

All blocks are prefixed by a four-byte field containing the size, which we will assume here is the actual size 1032 including the header and alignment rather than the user-available size of 1024. As the three low-order bits of the size are always zero, one of these bits is used as a flag to indicate whether the block is allocated or free. Crucially, another bit is used to indicate whether the *previous* block is allocated or free.

In addition to the size-plus-flags field, the first two 32-bit words of a free block are the forward and backward pointers linking the block into the doubly linked free-block list; the size-plus-flag field is also replicated as the last word of the block:



The strategy of the attacker, in brief, is to overwrite the `p` block in such a way that, when block `q` is freed, `malloc()` thinks block `p` is also free, and so attempts to coalesce them, in the process unlinking block `p`. But `p` was not in fact free, and the `forward` and `backward` pointers manipulated by `malloc()` as part of the unlinking process are in fact provided by the attacker. As we shall see, this allows writing two attacker-designated 32-bit words to two attacker-designated memory locations; if one of the locations holds a return address and is updated so as to point to attacker-supplied shellcode also in block `p`, the system has been compromised.

The normal operation of `free(q)`, for an arbitrary block `q`, includes the following:

- Get the block size (`size`) and flags at address `q-4`
- Check the following block at address `p+size` to see if it is free; if so, merge (we are not interested in this case)

- Check the flags to see if the *preceding* block is free; if so, load its size `prev_size` from address `q-8`, the address of the `copy of size` field in the diagram above; from that calculate a pointer to the previous block as `p = q - prev_size`; then unlink block `p` (as the coalesced block will go on a different free-block list).

For our specific block `q`, however, the attacker can overwrite the final `size` field of block `p`, `prev_size` above, and can also overwrite the flag at address `q-4` indicating whether or not block `p` is free. The attacker can *not* overwrite the header of block `p` that would properly indicate that block `p` was still in use, but the `free()` code did not double-check that.

We will suppose that the attacker has overwritten block `p` to include the following:

- setting the previous-block-is-free flag in the header of block `q` to true
- setting the final `size` field of block `p` to a desired value, `badsize`
- placing value `ADDR_F` at address `q-badsize`; this is where `free()` will believe the previous block's forward pointer is located
- placing the value `ADDR_B` at address `q-badsize+4`; this is where `free()` will believe the previous block's backward pointer is located

When the `free(q)` operation is now executed, the system will calculate the previous block as at address `p1 = q-badsize` and attempt to unlink the false "block" `p1`. The normal unlink is

```
(p->backward) -> forward  = p->forward;
(p->forward)  -> backward = p->backward;
```

Alas, when unlinking `p1` the result is

```
*ADDR_B        = ADDR_F
*(ADDR_F + 4)  = ADDR_B
```

where, for the pointer increment in the second line, we take the type of `ADDR_F` to be `char *` or `void *`.

At this point the jig is pretty much up. If we take `ADDR_B` to be the location of a return address on the stack, and `ADDR_F` to be the location of our shellcode, then the shellcode will be executed when the current stack frame returns. Extending this to a working example still requires a fair bit of attention to details; see *[AB03]*.

One important detail is that the data we use to overwrite block `p` generally cannot contain NUL bytes, and yet a small positive number for `badsize` will have several NUL bytes. One workaround is to have `badsize` be a small *negative* number, meaning that the false previous-block pointer `p1` will actually come after `q` in memory.

22.3.2 A JPEG heap vulnerability

In 2004 Microsoft released vulnerability notice MS04-028 (and patch), relating to a **heap buffer overflow** in **Windows XP SP1**. Microsoft had earlier provided a standard library for displaying JPEG images, known as GDI for Graphics Device Interface. If a specially formed JPEG image were opened via the GDI library, an overflow on the heap would occur that would launch a shellcode. While most browsers had their own, unaffected, JPEG-rendering subroutines, it was often not difficult to convince users to open a JPEG file supplied as an email attachment or via an html download link.

The problem occurred in the processing of the JPEG "comment" section, normally invisible to the user. The section begins with the flag 0xFFFE, followed by a two-byte length field (in network byte order) that was to include its own two bytes in the total length. During the image-file loading, 2 was subtracted from the length to get the calculated length of the actual data. If the length field had contained 1, for example, the calculated length would be -1, which would be interpreted as the unsigned value 2^{32} - 1. The library, thinking it had that amount of space, would then blithely attempt to write that many bytes of comment into the next heap page, overflowing it. These bytes in fact would come from the image portion of the JPEG file; the attacker would place here a NOPslide, some shellcode, and, as we shall see, a few other carefully constructed values.

As with the Linux heap described above in *22.3.1 A Linux heap vulnerability*, blocks on the WinXP heap formed a doubly-linked list, with forward and backward pointers known as `flink` and `blink`. As before, the allocator will attempt to unlink a block via

```
blink -> forward  = flink;
flink -> backward = blink;
```

The first of these simplifies to `*blink = flink`, as the offset to field `forward` is 0; this action allows the attacker to write any word of memory (at address `blink`) with any desired value.

The JPEG-comment-overflow operation eventually runs out of heap and results in a segmentation fault, at which point the heap manager attempts to allocate more blocks. However, the free list has already been overwritten, and so, as above, this block-allocation attempt instead executes `*blink = flink`.

The attacker's conceptual goal is to have `flink` hold an instruction that branches to the shellcode, and `blink` hold the address of an instruction that will soon be executed, or, equivalently, to have `flink` hold the address of the shellcode and `blink` represent a location soon to be loaded and used as the target of a branch. The catch is that the attacker doesn't exactly know where the heap is, so a variant of the return-to-libc approach described in *22.2.1 Return to libc* is necessary. The strategy described in the remainder of this section, described in *[JA05]*, is one of several possible approaches.

In Windows XP SP1, location `0x77ed73b4` holds a pointer to the entry point of the *Undefined Exception Filter* handler; if an otherwise-unhandled program exception occurs, Windows creates an `EXCEPTION_POINTERS` structure and branches to the address stored here. It is this address, which we will refer to as **UEF**, the attack will overwrite, by setting `blink = UEF`. A call to the Undefined Exception Filter will be triggered by a suitable subsequent program crash.

When the exception occurs (typically by the second operation above, `flink -> backward = blink`), before the branch to the address loaded from UEF, the `EXCEPTION_POINTERS` structure is created on the heap, overwriting part of the JPEG comment buffer. The address of this structure is stored in register `edi`.

It turns out that, scattered among some standard libraries, there are half a dozen instructions at known addresses of the form `call DWORD [edi+0x74]`, that is, "call the subroutine at 32-bit address `edi` + 0x74" (*[AHLR07]*, p 186). All these `call` instructions are intended for contexts in which register `edi` has been set up by immediately preceding instructions to point to something appropriate. In our attacker's context, however, `edi` points to the `EXCEPTION_POINTERS` structure; 0x74 bytes past `edi` is part of the attacker's overflowed JPEG buffer that is safely past this structure and will not have been overwritten by it. One such call instruction happens to be at address `0x77d92a34`, in `user32.dll`. This address is the value the attacker will put in `flink`.

So now, when the exception comes, control branches to the address stored in UEF. This address now points to the above `call DWORD [edi+0x74]`, so control branches again, this time into the attacker-controlled buffer. At this point, the processor lands on the NOPslide and ends up executing the shellcode (sometimes one more short jump instruction is needed, depending on layout).

This attack depends on the fact that a specific instruction, `call DWORD [edi+0x74]`, can be found at a specific, fixed address, `0x77d92a34`. Address-space layout randomization (*22.2.3.2 ASLR*) would have prevented this; it was introduced by Microsoft in Windows Vista in 2007.

22.3.3 Cross-Site Scripting (XSS)

In its simplest form, cross-site scripting involves the attacker posting malicious javascript on a third-party website that allows user-posted content; this javascript is then executed by the target computer when the victim visits that website. The attacker might leave a comment on the website of the following form:

```
I agree with the previous poster completely
<script> do_something_bad() </script>
```

Unless the website in question does careful html filtering of what users upload, any other site visitor who so much as *views* this comment will have the `do_something_bad()` script executed by his or her browser. The script might email information about the target user to the attacker, or might attempt to exploit a browser vulnerability on the target system in order to take it over completely. The script and its enclosing tags will not appear in what the victim actually sees on the screen.

The `do_something_bad()` code block will usually include javascript function definitions as well as function calls.

In general, the attacker can achieve the same effect if the victim visits the attacker's website. However, the popularity (and apparent safety) of the third-party website is usually important in practice; it is also common for the attack to involve obtaining private information from the victim's account on that website.

22.3.4 SQL Injection

SQL is the close-to-universal query language for databases; in a SQL-injection attack the attacker places a carefully chosen SQL fragment into a website form, in such a way that it gets executed. Websites typically construct SQL queries based on form data; the attacker's goal is to have his or her data treated as additional SQL. This is *supposed* to be prevented by careful quoting, but quoting often ends up not quite careful *enough*. A successful attacker has managed to run SQL code on the server that gives him or her unintended privileges or information.

As an example, suppose that the form has two fields, `username` and `password`. The system then runs the following sub-query that returns an empty set of records if the ⟨username,password⟩ pair is not found; otherwise the user is considered to be authenticated:

> select * from PASSWORDS p
> where p.user = '*username*' and p.password = '*password*';

The strings *username* and *password* are taken from the web form and spliced in; note that each is enclosed in single quotation marks *supplied by the server*. The attacker's goal is to supply username/password values so that a nonempty set of records will be returned, thus authenticating the attacker. The following values are successful here, where the quotation marks in *username* are supplied by the attacker:

> *username*: `'  OR 1=1 OR 'x'='y`
>
> *password*: `foo`

The spliced-together query built by the server is now

```
select * from PASSWORDS p
where p.user = '' OR 1=1 OR 'x'='y' and p.password = 'foo';
```

Note that of the eight single-quote marks in the where-clause, four (the first, sixth, seventh and eighth) came from the server, and four (the second through fifth) came from the attacker.

The where-clause here appears to SQL to be the disjunction of three OR clauses (the last of which is `'x'='y' and p.password = 'foo'`). The middle OR clause is `1=1` which is always true. Therefore, the login succeeds.

For this attack to work, the attacker must have a pretty good idea what query the server builds from the user input. There are two things working in the attacker's favor here: first, these queries are often relatively standard, and second, the attacker can often discover a great deal from the error messages returned for malformed entries. In many cases, these error messages even reveal the table names used.

See also xkcd.com/327.

22.4 Network Intrusion Detection

The idea behind **network intrusion detection** is to monitor one's network for signs of attack. Many newer network intrusion-detection systems (NIDS) also attempt to halt the attack, but the importance of simple monitoring and reporting should not be underestimated. Many attacks (such as password guessing, or buffer overflows in the presence of ASLR) take multiple (thousands or millions) of tries to succeed, and a NIDS can give fair warning.

There are also host-based intrusion-detection systems (HIDS) that run on and monitor a specific host; we will not consider these further.

Most NIDS detect intrusions based either on **traffic anomalies** or on pattern-based **signatures**. As an example of the former, a few pings (*7.11 Internet Control Message Protocol*) might be acceptable but a large number, or a modest number addressed to nonexistent hosts, might be cause for concern.

As for signatures, the attack in *22.3.2 A JPEG heap vulnerability* can be identified by the hex strings 0xFFFE0000 or 0xFFFE0001. What about the attack in *22.2.2.3 The exploit*? Using the shellcode itself as signature tends to be ineffective as shellcode is easy to vary. The NOPslide, too, can be replaced with a wide variety of other instructions that just happen to do nothing in the present context, such as `sub eax,eax`. One of the most common signatures used by NIDSs for overflow attacks is simply the presence of overly long strings; the false-positive rate is relatively low. In general, however, coming up with sufficiently specific signatures can be nontrivial. An attack that keeps changing in relatively trivial ways to avoid signature-based detection is sometimes said to be **polymorphic**.

22.4.1 Evasion

The NIDS will have to reassemble TCP streams (and sometimes sequences of UDP packets) in order to match signatures. This raises the possibility of **evasion**: the attacker might arrange the packets so that the NIDS reassembles them one way and the target another way. The possibilities for evasion are explored in great detail in *[PN98]*; see also *[SP03]*.

One simple way to do this is with overlapping TCP segments. What happens if one packet contains bytes 1-6 of a TCP connection as "help" and a second packet contains bytes 2-7 as "armful"?

```
h   e   l   p
    a   r   m   f   u   l
```

These can be reassembled as either "helpful" or "harmful"; the TCP specification does not say which is preferred and different operating systems routinely reassemble these in different ways. If the NIDS reassembles the packets one way, and the target the other, the attack goes undetected. If the attack is spread over multiple packets, there may be many more than two ways that reassembly can occur, increasing the probability that the NIDS and the target will differ.

Another possibility is that one of the overlapping segments has a header irregularity (in either the IP or TCP header) that causes it to be rejected by the target but not by the NIDS, or vice-versa. If the packets are

```
h   e   l   p
h   a   r   m   f   u   l
```

and both systems normally prefer data from the first segment received, then both would reassemble as "helpful". But if the first packet is rejected by the target due to a header flaw, then the target receives "harmful". If the flaw is not recognized by the NIDS, the NIDS does not detect the problem.

A very similar problem occurs with IPv4 fragment reassembly, although IPv4 fragments are at this point intrinsically suspicious.

One approach to preventing evasion is to configure the NIDS with information about how the actual target does its reassembly, so the NIDS can match it. An even safer approach is to have the NIDS reassemble any overlapping packets and then forward the result on to the potential target.

22.5 Cryptographic Goals

For the remainder of this chapter we turn to the use of cryptographic techniques in networking, to protect packet contents. Different techniques address different issues; three classic goals are the following:

1. Message **confidentiality**: eavesdroppers should not be able to read the contents.
2. Message **integrity**: the recipient should be able to verify that the message was received correctly, even in the face of a determined adversary along the way.
3. Sender **authentication**: the recipient should be able to verify the identity of the sender.

Briefly, confidentiality is addressed through encryption (*22.7 Shared-Key Encryption* and *22.9 Public-Key Encryption*), integrity is addressed through secure hashes (*22.6 Secure Hashes*), and authentication is addressed through secure hashes and public-key signatures.

Encryption by itself does not ensure message integrity. In *22.7.4 Stream Ciphers* we give an example using the message "Transfer $ **02**000 to Mallory". It is encrypted by XORing with the corresponding number of bytes of the keystream (*22.7 Shared-Key Encryption*), and decrypted by XORing again. If the attacker XORs the two bytes in bold with the byte ('0' XOR '2'), the message becomes "Transfer $ **20**000 to Mallory"; if the attacker XORs those bytes in the encrypted message with ('0' XOR '2') then the result will decrypt to the $20,000 transfer.

Similarly, integrity does not automatically ensure authentication. Two parties can negotiate a temporary key to guarantee message integrity without ever establishing each other's identities as is necessary for authentication. For example, if Alice connects to a website using SSL/TLS, but the site never purchased an SSL certificate (*22.10.2 TLS* and *22.10.2.1 Certificate Authorities*), or Alice does not trust the site's certificate authority (*22.10.2.1 Certificate Authorities*), then Alice has message integrity for her session, but not authentication.

To the above list we might add resistance to message **replay**. Consider messages such as the following:

1. "I, Alice, authorize the payment to Bob of $1000 from my account"
2. My facebook.com authentication cookie is Zg8yPCDwbzJ-59Hc-DvHt67qrS

Alice does not want the first message to be executed by her bank more than once, though doing so would violate none of the three rules above. The owner of the second message might be happy to replay it multiple times, but would not want someone *else* to be able to do so. One straightforward way to prevent replay attacks is to introduce a message timestamp or count.

We might also desire resistance to **denial-of-service attacks**. Such attacks are generally related to implementation failures. Attacks in which SSL users end up negotiating a very early version of the protocol, and thus a much less secure encryption mechanism, are arguably of this type; see the POODLE sidebar at *22.10 SSH and TLS*.

Finally, one sometimes sees *message non-repudiation* as a goal. However, in the technical – as opposed to legal – realm we can never hope to prove Alice herself signed a message; the best we can do is to prove that Alice's *key* was used to sign the message. At this point non-repudiation is, for our purposes here, quite similar to authentication. See, however, the final example of *22.6.1 Secure Hashes and Authentication*.

22.5.1 Alice and Bob

Cryptographers usually use Alice and Bob, instead of A and B, as the names of the parties sending each other encrypted messages. This tradition dates back at least to *[RSA78]*. Other parties sometimes include Eve the eavesdropper and Mallory the active attacker (and launcher of man-in-the-middle attacks). (In this article the names Aodh and Bea are introduced, though not specifically for cryptography; the Irish name Aodh is pronounced, roughly, as "Eh".)

22.6 Secure Hashes

How can we tell if a file has been changed? One way is to create a record of the file's checksum, using, for the sake of argument, the Internet checksum of *5.4 Error Detection*. If the potential file corruption is random, this will fail to detect a modified file with probability only 1 in 2^{16}.

Alas, if the modification is *intentional*, it is trivial to modify the file so that it has the same checksum as the original. If the original file has checksum c_1, and after the initial modification the checksum is now c_2, then all we have to do to create a file with checksum c_1 is to append the 16-bit quantity $c_2 - c_1$ (where the subtraction is done using ones-complement; we need to modify this slightly if the number of bytes in the first-draft modified file is odd). The CRC family of checksums is almost as easily tricked.

The goal of a **cryptographically secure hash** is to provide a hash function – we will not call it a "checksum" as hashes based on simple sums all share the insecurity above – for which this trick is well-nigh impossible. Specifically, we want a hash function such that:

- Knowing the hash value should shed no practical light on the message
- Given a hash value, there should be no feasible way to find a message yielding that hash

A slightly simpler problem than the second one above is to find two messages that have the same hash; this is sometimes called the **collision problem**. When the collision problem for a hash function has been solved, it is (high) time to abandon it as potentially no longer secure.

If a single bit of the input is changed, the secure-hash-function output is usually entirely different.

Hashes popular in the 1990s were the 128-bit MD5 (**RFC 1321**, based on MD4, *[RR91]*) and the 160-bit SHA-1 (developed by the NSA); SNMPv3 (*21.15 SNMPv3*) originally supported both of these (*21.15.2 Cryptographic Fundamentals*). MD5 collisions (two messages hashing to the same value) were reported in 2004, and collisions where both messages were meaningful were reported in 2005; such **collision attacks** mean it can no longer be trusted for security purposes.

Hash functions currently (2014) believed secure include the SHA-2 family, which includes variants ranging from 224 bits to 512 bits (and known individually as SHA-224 through SHA-512).

A common framework for constructing n-bit secure-hash functions is the Merkle-Dåmgard construction (*[RM88]*, *[ID89]*); it is used by MD5, SHA-1 and SHA-2. The initial n-bit state is specified. To this is then applied a set of transformations $H_i(x,y)$ that each take an n-bit block x and some additional bits y and return an updated n-bit block. These transformations are usually similar to the **rounds functions** of a block cipher; see *22.7.2 Block Ciphers* for a typical example. In encryption, the parameter y would be used to hold the key, or a substring of the key; in secure-hash functions, the parameter y holds a substring of the input message. The set of transformations is applied repeatedly as the process iterates over the entire input message; the result of the hash is the final n-bit state.

In the MD5 hash function, the input is processed in blocks of 64 bytes. Each block is divided into sixteen 32-bit words. One such block of input results in 64 iterations from a set of sixteen rounds-functions H_i, each applied four times in all. Each 32-bit input word is used as the "key" parameter to one of the H_i four times. If the total input length is not a multiple of 512 bits, it is padded; the padding includes a length attribute so two messages differing only by the amount of padding should not hash to the same value.

While this framework in general is believed secure, and is also used by the SHA-2 family, it does suffer from what is sometimes called the **length-extension** vulnerability. If h = hash(m), then the value h is simply the final state after the above mechanism has processed message m. An attacker knowing only h can then initialize the above algorithm with h, and continue it to find the hash $h' = \text{hash}(m^\frown m')$, for an arbitrary message m' concatenated to the end of m, *without knowing m*. If the original message m was padded to message m_p, then the attacker will find $h' = \text{hash}(m_p{}^\frown m')$, but that is often enough. This vulnerability must be considered when using secure-hash functions for message authentication, below.

The SHA-3 family of hash functions does not use the Merkle-Dåmgard construction and is believed not vulnerable to length-extension attacks.

22.6.1 Secure Hashes and Authentication

Secure hash functions can be used to implement message authentication. Suppose the sender and receiver share a secret, pre-arranged "key", K, a random string of length comparable to the output of the hash. Then, in principle, the sender can append to the message m the value h = hash(K⁀m). The receiver, knowing K, can recalculate this value and verify that the h appended to the message is correct. In theory, only someone who knew K could calculate h.

This *particular* hash(K⁀m) implementation is undermined by the length-extension vulnerability of the previous section. If the hash function exhibits this vulnerability and the sender transmits message m together with hash(K⁀m), then an attacker can modify this to message $m^\frown m'$ together with $\text{hash}(K^\frown m^\frown m')$, *without knowing K*.

This problem can be defeated by reversing the order and using hash(m⁀K), but this now introduces potential collision vulnerabilities: if the hash function has the length-extension vulnerability and two messages m_1 and m_2 hash to the same value, then so will $m_1{}^\frown K$ and $m_2{}^\frown K$.

Taking these vulnerabilities into account, RFC 2104 defines the **Hash Message Authentication Code**, or HMAC, as follows; it can be used with any secure-hash function whether or not it has the length-extension vulnerability. The 64-byte length here comes from the typical input-block length of many secure-hash functions; it may be increased as necessary. (SHA-512, of the SHA-2 family, has a 128-byte input-block length and would be a candidate for such an increase.)

- The shared key K is extended to 64 bytes by padding with zeroes.
- A constant string **ipad** is formed by repeating the octet 0x36 (0011 0110) 64 times.
- A constant string **opad** is formed by repeating 0x5c (0101 1100) 64 times.
- We set **K1** = K XOR **ipad**.
- We set **K2** = K XOR **opad**.

Finally, the HMAC value is

HMAC = hash(**K2** ⁀ hash(**K1**⁀**mesg**))

The values 0x36 (0011 0110) and 0x5c (0101 1100) are not critical, but the XOR of them has, by intent, a reasonable number of both 0-bits and 1-bits; see *[BCK96]*.

The HMAC algorithm is, somewhat surprisingly, believed to be secure even when the underlying hash function is vulnerable to some kinds of collision attacks; see *[MB06]* and RFC 6151. That said, a hash function vulnerable to collision attacks may have other vulnerabilities as well, and so HMAC-MD5 should still be phased out.

Negotiating the pre-arranged key K can be a significant obstacle, just as it can be for ciphers using pre-arranged keys (*22.7 Shared-Key Encryption*). If Alice and Bob meet in person to negotiate the key K, then Alice can use HMAC for authentication of Bob, as long as K is not compromised: if Alice receives a message signed with K, she knows it must have come from Bob.

Sometimes, however, K is negotiated on a per-session basis, without definitive "personal" authentication of the other party. This is akin to Alice and someone claiming to be "Bob" selecting an encryption key using the Diffie-Hellman-Merkle mechanism (*22.8 Diffie-Hellman-Merkle Exchange*); such key selection is secure and does not require that Alice or Bob have any prior knowledge of one another. In the HMAC setting, Alice can be confident that any HMAC-signed message was sent by the same "Bob" that negotiated the key, and not by a third party (assuming neither side has leaked the key K). This is true even if Alice is not sure "Bob" is the real Bob, or has no idea who "Bob" might be. The signature guarantees the message *integrity*, but also serves as *authentication* that the sender is the same entity who sent the previous messages in the series.

Finally, if Alice receives a message from Bob signed with HMAC using a pre-arranged secret key K (*22.6.1 Secure Hashes and Authentication*), Alice may herself trust the signature, but she cannot prove to anyone else that K was used to sign the message without divulging K. She also cannot prove to anyone else that Bob is the only other party who knows K. Therefore this signature mechanism provides authentication but not non-repudiation.

22.6.2 Password Hashes

A site generally does not store user passwords directly; instead, a hash of the password, h(p), is stored. The cryptographic hash functions above, *eg* MD5 and SHA-n, work well as long as the password file itself is not compromised. However, these functions all execute very quickly – by design – and so an attacker who succeeds in obtaining the password file can usually extract passwords simply by calculating the hash of every possible password. There are about 2^{14} six-letter English words, and so there are about 2^{38} passwords consisting of two such words and three digits. Such passwords are usually considered rather strong, but brute-force calculation of h(p) for 2^{38} possible values of p is quite feasible for the hashes considered so far. Checking 10^8 MD5-hashed passwords per second is quite feasible; this means $2^{38} = 256 \times 2^{30} \simeq 256 \times 10^9$ passwords can be checked in under 45 minutes. Use of a GPU increases the speed at least 10-fold.

Special hashes have been developed to address this. Two well-known ones are scrypt (*[CP09]* and this Internet Draft) and bcrypt. A newer entrant is Argon2, while PBKDF2 is an old stalwart. The goal here is to develop a hash that takes – ideally – the better part of a second to calculate even once, and which cannot easily be sped up by large precomputed tables. Typically these are something like a thousand to a million times slower than conventional hash functions like MD5 and the SHA-2 family; a millionfold-slower hash function is the equivalent of making the passwords longer by 20 random bits ($10^6 \simeq 2^{20}$). See *[ACPRT16]* for a theoretical analysis of the effectiveness of scrypt in this context. See also **RFC 2898**, section 4.2, though the proposal there is only a thousandfold slower.

(Good password tables also incorporate a *salt*, or *nonce*: a random string s is chosen and appended to the password before hashing; the password record is stored as the pair $\langle$s,h(p$\frown$s)$\rangle$. This means that cracking the password for one user will not expose a second user who has chosen exactly the same password, so long as the second user has a different salt s. Salting, however, does not change the scenarios outlined above.)

22.6.3 CHAP

Secure hashes can also be used to provide secure password-based login authentication in the presence of eavesdropping. Specific implementations include CHAP, the Challenge-Handshake Authentication Protocol (**RFC 1994**) and Microsoft's MS-CHAP (version 1 in **RFC 2433** and version 2 in **RFC 2759**). The general idea is that the server sends a random string (the "challenge") and the client then creates a response consisting of a secure hash of the challenge string concatenated with the user's password (and possibly other information). Assuming the secure hash is actually secure, only someone in possession of the user password could have created this response. By the same token, an eavesdropper cannot figure out the password from the response.

While such protocols can be quite secure in terms of verifying to the server that the client knows the password, the CHAP strategy has a fundamental vulnerability: the server must store the plaintext password rather than a secure hash of it (as in the previous section). An attacker who makes off with the server's password file then has everything; no brute-force password cracking is needed. For this reason, newer authentication protocols often will create an encrypted channel first (*eg* using TLS, *22.10.2 TLS*), and then use that secure channel to exchange credentials. This may involve transmission of the *plaintext* password, which clearly allows the server to store only hashed passwords. However, as the next example shows, it *is* possible to authenticate without having the server store plaintext passwords and without having the plaintext password transmitted at all.

22.6.4 SCRAM

SCRAM (Salted Challenge-Response Authentication Mechanism), **RFC 5802**, is an authentication protocol with the following features:

- The client password is not transmitted in the clear
- The server does not store the plaintext client password
- If the server's hashed credentials are compromised, an attacker still cannot authenticate

We outline a very stripped-down version of the protocol: password salting has been removed for clarity, and the exchange itself has been greatly simplified. The **ClientKey** is a hashed version of the password; what the server stores is a secure hash of the ClientKey called **StoredKey**:

```
StoredKey = hash(ClientKey)
```

The exchange begins with the server sending a random **nonce** string to the client. The client now calculates the **ClientSignature** as follows:

```
ClientSignature = hash(StoredKey, nonce)
```

Only an attacker who has eavesdropped on this exchange can replicate the ClientSignature, but the server can compute it. The client then sends the following to the server:

```
ClientKey XOR ClientSignature
```

Because the server can calculate ClientSignature, it can extract ClientKey, and verify that h(ClientKey) = StoredKey. An attacker who has obtained StoredKey from the server can generate ClientSignature, but cannot generate ClientKey.

However, an attacker who has *both* exfiltrated StoredKey from the server *and* eavesdropped on a client-server exchange is able to generate the ClientSignature and thus extract the ClientKey. The attacker can then authenticate at a later time using ClientKey. For this reason, a secure tunnel is still needed for the authentication. Given the presence of such a tunnel, the SCRAM approach appears to offer a relatively modest security improvement over sending the password as plaintext. Note, though, that with SCRAM the server does not see the password at all, so if the client mistakenly connects to the wrong server, the password is not revealed.

See below at *22.8.2 Simultaneous Authentication of Equals* for another, mutual, form of password authentication.

22.7 Shared-Key Encryption

Secure hashes can provide authentication, but to prevent eavesdropping we need encryption. While **public-key** encryption (*22.9 Public-Key Encryption*) is perhaps more glamorous, the workhorse of the encryption world is the **shared-key cipher**, or **shared-secret** or **symmetric** cipher, in which each party has possession of a key K, used for both encryption and decryption.

Shared-key ciphers are often quite fast; public-key encryption is generally slower by several orders of magnitude. As a result, if two parties without a shared key wish to communicate, they will almost always used public-key encryption only to exchange a key for a shared-key cipher, which will then be used for the actual message encryption.

Typical key lengths for shared-key ciphers believed secure range from 128 bits to 256 bits. For most shared-key ciphers there are no known attacks that are much faster than brute force, and $2^{256} \simeq 10^{77}$ is quite a large number.

Shared-key ciphers can be either **block ciphers**, encrypting data in units of blocks that might typically be 8 bytes long, or **stream** ciphers, which generate a pseudorandom **keystream**. Each byte (or even bit) of the message is then encrypted by (typically) XORing it with the corresponding byte of the keystream.

22.7.1 Session Keys

The more messages a key is used to encrypt, the more information a potential eavesdropper may have with which to launch a codebreaking attack. If Alice and Bob have a pre-arranged shared secret key K, is therefore quite common for them to encrypt the bulk of their correspondence with temporary **session keys**, each one used for a time period that may range from minutes to days. The secret key K might then be used only to exchange new session keys and to forestall man-in-the-middle attacks (*22.9.3 Trust and the Man in the Middle*) by signing important messages (*22.6.1 Secure Hashes and Authentication*). If Diffie-Hellman-Merkle key exchange (ref:*diffie-hellman*) is used, K might be reserved *only* for message signing.

Sometimes session keys are entirely different keys, chosen randomly; if such a key is discovered, the attacker learns nothing about the secret key K. Other times, the session key is a mash-up of K and some additional information (such as an initialization vector, *22.7.3 Cipher Modes*), that is transmitted in the clear. The session key might then be the concatenation of K and the initialization vector, or the XOR of the two. Either approach means that an eavesdropper who figures out the session key also has the secret key K, but the use of such session keys still may make any codebreaking attempt much more difficult.

In *22.9.2 Forward Secrecy* we consider the reverse problem of how Alice and Bob might keep a session key private even if the secret key is compromised.

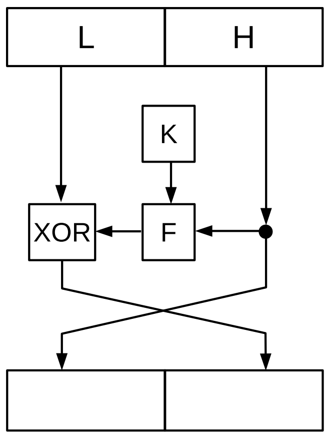
22.7.2 Block Ciphers

As mentioned, a block cipher encrypts data one block at a time, typically with a key length rather longer than the block size. A typical block cipher proceeds by the iterated application of a sequence of **round functions**, each updating the block and using some round-dependent substring of the key as auxiliary input. If we start with a block of plaintext P and apply all the round functions in turn, the result is the ciphertext block C = E(P,K) = E_K(P). The process can be reversed so that P = D(C,K) = D_K(C).

A common framework is that of the **Feistel network**. In such an arrangement, the block is divided into two or more words sized for the machine architecture; an 8-byte block is typically divided into two 32-bit words which we can call L and H for Low and High. A typical round function is now of the following form, where K is the key and F(x,K) takes a word x and the key K and returns a new word:

$\langle$L,H$\rangle \longrightarrow \langle$H, L XOR F(H,K)$\rangle$

Visually, this is often diagrammed as



One round here scrambles only half the block, but the other half gets scrambled in the next round (sometimes the operation above is called a half-round for that reason). The total number of rounds is typically somewhere between 10 and 50. Much of the art of designing high-quality block ciphers is to come up with round functions that result in overall very fast execution, without introducing vulnerabilities. The more rounds, the slower.

The internal function F, often different for each round, may look at only a designated subset of the bits of the key K. Note that the operation above is invertible – that is, can be decrypted – regardless of F; given the right-hand side the receiver can compute F(H,K) and thus, by XORing this with L XOR F(H,K), can recover L. This remains true if, as is sometimes the case, the XOR operation is replaced with ordinary addition.

Crypto Law

The Salsa20 cipher mentioned here is a member of Daniel Bernstein's "snuffle" family of ciphers based on secure-hash functions. In the previous century, the US government banned the export of ciphers but not secure-hash functions. They also at one point banned the export (and thus publication) of one of Bernstein's papers; he sued. In 1999, the US Court of Appeals for the Ninth Circuit found in his favor. The decision, 176 F.3d 1132, is the only appellate decision of which the author is aware that contains (in the footnotes) not only samples of C code, but also of Lisp.

A simple F might return the result of XORing H and a subset of the bits of K. This is usually a little *too* simple, however. Ordinary modulo-32 addition of H and a subset of K often works well; the interaction of addition and XORing introduces considerable bit-shuffling (or *diffusion* in the language of cryptography). Other operations used in F(H,K) include Boolean AND and OR operations. 32-bit multiplication introduces considerable bit-shuffling, but is often computationally more expensive. The **Salsa20** cipher of *[DB08]* uses only XOR and addition, for speed.

It is not uncommon for the round function also to incorporate "bit rotation" of one or both of L and H; the result of bit-rotation of 1000 **111**0 two places to the left is 00**11 1**010.

If a larger blocksize is desired, say 128 bits, but we still want to avail ourselves of the efficiency of 32-bit operations, the block can be divided into ⟨A,B,C,D⟩. The round function might then become

$\langle A,B,C,D\rangle \longrightarrow \langle B,C,D, (A \text{ XOR } F(B,C,D,K)) \rangle$

As mentioned above, many secure-hash functions use block-cipher round functions that then use successive chunks of the message being hashed in place of the key. In the MD5 algorithm, block A above is transformed into the 32-bit sum of input-message fragment M, a constant K, and G(B,C,D) which can be any of several Boolean combinations of B, C and D.

An alternative to the Feistel-network framework for block ciphers is the use of so-called substitution-permutation networks.

The first block cipher in widespread use was the federally sanctioned Data Encryption Standard, or **DES** (commonly pronounced "dez"). It was developed at IBM by 1974 and then selected by the US National Bureau of Standards (NBS) as a US standard after some alterations recommended by the National Security Agency (NSA). One of the NSA's recommendations was that a key size of 56 bits was sufficient; this was in an era when the US government was very concerned about the spread of strong encryption. For years, many people assumed the NSA had intentionally introduced other weaknesses in DES to facilitate government eavesdropping, but after forty years no such vulnerability has been discovered and this no longer seems so likely. The suspicion that the NSA had in the 1970's the resources to launch brute-force attacks against DES, however, has greater credence.

In 1997 an academic team launched a successful brute-force attack on DES. The following year the Electronic Frontier Foundation (EFF) built a hardware DES cracker for about US$250,000 that could break DES in a few days.

In an attempt to improve the security of DES, triple-DES or **3DES** was introduced. It did not become an official standard until the late 1990's, but a two-key form was proposed in 1978. 3DES involves three applications of DES with keys ⟨K1,K2,K3⟩; encrypting a plaintext P to ciphertext C is done by $C = E_{K3}(D_{K2}(E_{K1}(P)))$. The middle deciphering option D_{K2} means the algorithm reduces to DES when K1 = K2 = K3; it also reduces exposure to a particular vulnerability known as "meet in the middle" (no relation to "man in the middle"). In *[MH81]* it is estimated that 3DES with three distinct keys has a strength roughly equivalent to 2×56 = 112 bits. That same paper also uses the meet-in-the-middle attack to show that straightforward "double-DES" encryption C = $E_{K2}(E_{K1}(P))$ has an effective keystrength of only 56 bits – no better than single DES – if sufficient quantities of plaintext and corresponding ciphertext are known.

As concerns about the security of DES continued to mount, the US National Institute of Standards and Technology (NIST, the new name for the NBS) began a search for a replacement. The result was the Advanced Encryption Standard, **AES**, officially approved in 2001. AES works with key lengths of 128, 192 and 256 bits. The algorithm is described in *[DR02]*, and is based on the Rijndael family of ciphers; the name Rijndael ("rain-dahl") is a combination of the authors' names.

Earlier non-DES ciphers include **IDEA**, the International Data Encryption Algorithm, described in *[LM91]*, and **Blowfish**, described in *[BS93]*. Both use 128-bit keys. The IDEA algorithm was patented; Blowfish was intentionally not patented. A successor to Blowfish is Twofish.

22.7.3 Cipher Modes

The simplest encryption "mode" for a block cipher is to encrypt each input block independently. That is, if P_i is the ith plaintext block, then the ith ciphertext block C_i is $E(P_i,K)$. This is known as electronic codebook or **ECB** mode.

ECB is vulnerable to known-plaintext attacks. Suppose the attacker knows that the message is of the following form, where the vertical lines are the block boundaries:

```
Bank Transfer   |   to MALLORY   |   Amount $1000
```

If the attacker also knows the ciphertext for "Amount $100,000", and is in a position to rewrite the message (or to inject a new message), then the attacker can combine the first two blocks above with the third $100,000 block to create a rather different message, without knowing the key. At http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation there is a remarkable example of the failure of ECB to fail to effectively conceal an encrypted image.

As a result, ECB is seldom used. A common alternative is cipher block chaining or **CBC** mode. In this mode, each plaintext block is XORed with the previous ciphertext block before encrypting:

$$C_i = E(K,C_{i-1} \text{ XOR } P_i)$$

To decrypt, we use

$$P_i = D(K,C_i) \text{ XOR } C_{i-1}$$

If we stop here, this means that if two messages begin with several identical plaintext blocks, the encrypted messages will also begin with identical blocks. To prevent this, the first ciphertext block C_0 is a random string, known as the **initialization vector**, or IV. The plaintext is then taken to start with block P_1. The IV is sent in the clear, but contains no private information.

See exercise 5.0.

22.7.4 Stream Ciphers

Conceptually, a stream cipher encodes one byte at a time. The cipher generates a pseudorandom **keystream**. If K_i is the ith byte of the keystream, then the ith plaintext byte P_i is encrypted as $C_i = P_i$ XOR K_i. A stream cipher can be viewed as a special form of pseudorandom number generator or PRNG, though PRNGs used for numeric methods and simulations are seldom secure enough for cryptography.

Simple XORing of the plaintext with the keystream may not seem secure, but if the keystream is in fact truly random then this cipher is unbreakable: to an attacker, all possible plaintexts are equally likely. A truly random keystream means that the entire keystream must be shared ahead of time, and no portion may ever be reused; this cipher is known as the **one-time pad**.

Stream ciphers do not, by themselves, provide authenticity. An attacker can XOR something with the encrypted message to change it. For example, if the message is known to be

```
Transfer $ 02000 to Mallory
           ^^
```

then the attacker can XOR the two bytes over the "^" with '0' XOR '2', changing the character '0' to a '2' and the '2' to a '0'. The attacker does this to the encrypted stream, but the decrypted plaintext stream retains the change. Appending an authentication code such as HMAC, *22.6.1 Secure Hashes and Authentication*, prevents this.

Stream ciphers are, in theory, well suited to the encryption of data sent a single byte at a time, such as data from a telnet session. The ssh protocol (*22.10.1 SSH*), however, generally uses block ciphers; when it has to send a single byte it pads the block with random data.

22.7.4.1 RC4

The RC4 stream cipher was quite widely used. It was developed by Ron Rivest at RSA Security in 1987, but never formally published. The code was leaked, however, and so the internal details are widely available. "Unofficial" implementations are sometimes called **ARC4** or ARCFOUR, the "A" for "Alleged".

RC4 was designed for very fast implementation in software; to this end, all operations are on whole bytes. RC4 generates a keystream of pseudorandom bytes, each of which is XORed with the corresponding plaintext byte. The keystream is completely independent of the plaintext.

The key length can range from 5 up to 256 bytes. The unofficial protocol contains no explicit mechanism for incorporating an initialization vector, but an IV is well-nigh essential for stream ciphers; otherwise an identical keystream is generated each time. One simple approach is to create a session key by concatenating the IV and the secret RC4 key; alternatively (and perhaps more safely) the IV and the RC4 key can be XORed together.

RC4 was the basis for the ill-fated WEP Wi-Fi encryption, *22.7.7 Wi-Fi WEP Encryption Failure*, due in part to WEP's requirement that the 3-byte IV precede the 5-byte RC4 key. The flaw there did not affect other applications of RC4, but newer attacks have suggested that RC4 be phased out.

Internally, an RC4 implementation maintains the following state variables:

An array S[] representing a permutation i→S[i] of all bytes
two 8-bit integer indexes to S[], i and j

S[] is guaranteed to represent a permutation (*ie* is guaranteed to be a 1-1 map) because it is initialized to the identity and then all updates are transpositions (involving swapping S[i] and S[j]).

Initially, we set S[i] = i for all i, and then introduce 256 transpositions. This is known as the **key-scheduling algorithm**. In the following, `keylength` is the length of the key `key[]` in bytes.

```
J=0;
for I=0 to 255:
    J = J + S[I] + key[I mod keylength];
    swap S[I] and S[J]
```

As we will see in *22.7.7 Wi-Fi WEP Encryption Failure*, 256 transpositions is apparently not enough.

After initialization, `I` and `J` are reset to 0. Then, for each keystream byte needed, the following is executed, where `I` and `J` retain their values between calls:

```
I = (I+1) mod 256
J = J + S[I] mod 256
swap S[I] and S[J]
return S[ S[I] + S[J] mod 256 ]
```

For `I` = 1, the first byte returned is `S[ S[1] + S[S[1]] ]`.

22.7.5 Block-cipher-based stream ciphers

It is possible to create a stream cipher out of a block cipher. The first technique we will consider is **counter mode**, or CTR. The sender and receiver agree on an initial block, B, very similar to an initialization vector. The first block of the keystream, K_0, is simply the encrypted block E(B,K). The ith keystream block, for i>0, is K_i = E(B+i,K), where "B+i" is the result of treating B as a long number and performing ordinary arithmetic.

Going from B+1 to B+(i+1) typically changes only one or two bits, but no significant vulnerability based on this has ever been found, and this form of counter mode is now widely believed to be as secure as the underlying block cipher.

A related technique is to generate the keystream by starting with a secret key K and taking a secure hash of each of the successive concatenations $K\frown 1$, $K\frown 2$, *etc*. The drawback to this approach is that many secure-hash functions have not been developed with this use in mind, and unforeseen vulnerabilities may exist.

The keystream calculated by counter mode is completely independent of the plaintext. This is not always the case. In **cipher feedback** mode, or CFB, we initialize C_0 to the initialization vector, and then define the keystream blocks K_i for i>0 as follows:

$$K_i = E_K(C_{i-1})$$

As usual, $C_i = P_i$ XOR K_i for i>0, so the right-hand side above is $E_K(P_{i-1}$ XOR $K_{i-1})$. The evolution of the keystream thus depends on earlier plaintext. Note that, by the time we need to use K_i, the earlier plaintext P_{i-1} has been entirely received.

Stream ciphers face an error-recovery problem, but we will ignore that here as we will assume the transport layer provides sufficient assurances that the ciphertext is received accurately.

22.7.6 Encryption and Authentication

If the ciphertext is modified in transit, most decryption algorithms will happily "decrypt" it anyway, though perhaps to gibberish; encryption provides no implicit validity check. Worse, it is sometimes possible for an attacker to modify the ciphertext so as to produce a meaningful, intentional change in the resultant plaintext. Examples of this appear above in *22.7.4 Stream Ciphers* and *22.7.3 Cipher Modes*; for CBC block ciphers see exercise 5.0.

It is therefore common practice to include HMAC authentication signatures (*22.6.1 Secure Hashes and Authentication*) with each encrypted message; the use of HMAC rather than a simple checksum ensures that an attacker cannot modify messages and have them still appear to be valid. HMAC may not identify the sender as positively as public-key digital signatures, *22.9.1.1 RSA and Digital Signatures*, but it *does* positively identify the sender as the same entity with whom the receiver negotiated the key. This is all that is needed.

Appending an HMAC signature does more than prevent garbled messages; there are attacks that, in the absence of such signatures, allow full decryption of messages (particularly if the attacker can create multiple ciphertexts and find out which ones the recipient was able to decrypt successfully). See *[SV02]* for an example.

There are three options for combining encryption with HMAC; for consistency with the literature we replace HMAC with the more general MAC (for Message Authentication Code):

- MAC-then-encrypt: calculate the MAC signature on the plaintext, append it, and encrypt the whole
- encrypt-and-MAC: calculate the MAC signature on the plaintext, encrypt the message, and append to that the MAC
- encrypt-then-MAC: encrypt the plaintext and calculate the MAC of the ciphertext; append the MAC to the ciphertext

These are analyzed in *[BN00]*, in which it is proven that **encrypt-then-MAC** in general satisfies some stronger cryptographic properties than the others, although these properties may hold for the others in special cases. Encrypt-then-MAC means that no decryption is attempted of a forged or spoofed message.

22.7.7 Wi-Fi WEP Encryption Failure

The WEP (Wired-Equivalent Privacy) mechanism was the first Wi-Fi encryption option, introduced in 1999; see *3.7.5 Wi-Fi Security*. It used RC4 encryption with either a 5-byte or 13-byte secret key; this was always appended to a 3-byte initialization vector that, of necessity, was sent in the clear. The RC4 session key was thus either 8 or 16 bytes; the shorter key was more common.

There is nothing inherently wrong with that strategy, but the designers of WEP made some design choices that were, in retrospect, infelicitous:

- the secret key was appended to the IV (rather than, say, XORed with it)
- the IV was only three bytes long
- a new RC4 keystream and new IV was used for each packet

The third choice above was perhaps the more serious mistake. One justification for this choice, however, was that otherwise lost packets (which are common in Wi-Fi) would represent gaps in the received stream, and the receiver might have trouble figuring out the size of the gap and thus how to decrypt the later arrivals.

A contributing issue is that the first byte of the plaintext – a header byte from the Wi-Fi packet – is essentially known. This first byte is generally from the Logical Link Control header, and contains the same value as the Ethernet Type field (see *3.7.4 Access Points*). All IP packets share the same value. Thus, by looking at the first byte of the encrypted packet, an attacker knows the first byte of the keystream.

A theoretical WEP vulnerability was published in [FMS01] that proved all-too-practical in the real world. We will let K[] denote the 8-byte key, with K[0],K[1],K[2] representing the three-byte initialization vector and K[i] for $3 \leqslant i < 8$ representing the *secret* key. The attacker monitors the airwaves for IVs of a particular form; after about 60 of these are collected, the attacker can make an excellent guess at K[3]. Now a slightly different group of IVs is collected, allowing a guess at K[4] and so on. The bytes of the secret key thus fail sequentially. The attack requires about the same (modest) time for each key byte discovered, so a 16-byte total key length does not add much more security versus the 8-byte key.

Because the IV is only three bytes, the time needed to collect packets containing the necessary special IV values is modest.

We outline the attack on the first secret byte, K[3]. The IV's we are looking for have the form

$$\langle 3,-1,X\rangle$$

where -1 = 255 for 8-bit bytes; these are sometimes called **weak IVs**. One IV in 2^{16} is of this form, but a more careful analysis (which we omit) can increase the usable fraction of IVs substantially.

To see why these IVs are useful, we need to go back to the RC4 key-scheduling algorithm, *22.7.4.1 RC4*. We start with an example in which the first six bytes of the key is

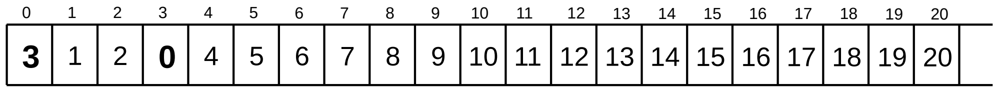
```
K[] = [3,-1,5,4,-14,3]
```

The IV here is $\langle 3,-1,5\rangle$.

Recall that the array S is initialized with S[i] = i; this represents S = S_0. At this point, I and J are also 0. We now run the following loop, introducing one transposition to S per iteration:

```
for I=0 to 255:
    J = J + S[I] + K[I mod keylength];
    swap S[I] and S[J]
```

The first value of `J`, when `I` = 0, is `K[0]` which is 3. After the first transposition, S is as follows, where the swapped values are in bold:



For the next iteration, `I` is 1 and `J` is 3+1+(-1) = 3 again. After the swap, S is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	**0**	2	**1**	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Next, `I` is 2 and `J` is 3+2+5 = 10. In general, if the IV were ⟨3,-1,X⟩, `J` would be 5+X.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	0	**10**	1	4	5	6	7	8	9	**2**	11	12	13	14	15	16	17	18	19	20

At this point, we have processed the three-byte IV; the next iteration involves the first secret byte of `K`. `I` is 3 and `J` becomes 10+1+`K[3]` = 15:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	0	10	**15**	4	5	6	7	8	9	2	11	12	13	14	**1**	16	17	18	19	20

Recall that the first byte returned in the keystream is `S[ S[1] + S[S[1]] ]`. At this point, that would be `S[0+3]` = 15.

From this value, 15, and the IV ⟨3,-1,5⟩, the attacker can calculate, by repeating the process above, that `K[3]` = 4.

If the IV were ⟨3,-1,X⟩, then, as noted above, in the `I`=2 step we would have `J` = 5+X, and then in the `I`=3 step we would have `J` = 6 + X + `K[3]`. If Y is the value of the first byte of the keystream, using `S[]` as of this point, then `K[3]` = Y – X – 6 (assuming that X is not -5 or -4, so that `S[0]` and `S[1]` are not changed in step 3).

If none of `S[0]`, `S[1]` and `S[3]]` are changed in the remaining 252 iterations, the value we predict here after three iterations – *eg* 15 – would be the first byte of the actual keystream. Treating future selections of the value `J` as random, the probability that one iteration does *not* select `J` in the set {0,1,3} – and thus leaves these values alone – is 253/256. The probability that the remaining iterations do not change these values in `S[]` is thus about $(253/256)^{252} \simeq 5\%$

A 5% success rate is not terribly impressive, on the face of it. But 5% of the time we identify `K[3]` correctly, and the other 95% of the time the (incorrect) values we calculate for `K[3]` are uniformly, randomly distributed in the range 0 to 255. With 60 guesses, we expect about three correct guesses. The other 57 are spread about with, most likely, no more than two guesses for any one value. If we look for the value guessed most often, it is likely to be the true value of `K[3]`; increasing the number of ⟨3,-1,X⟩ IVs will increase the certainty here.

For the sake of completeness, in the next two iterations, `I` is 4 and 5 respectively, and `J` is 15+4+(-14) = 5 and 5+4+3=12 respectively. The corresponding contents of `S[]` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	0	10	15	**5**	**4**	6	7	8	9	2	11	12	13	14	1	16	17	18	19	20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	0	10	15	5	**12**	6	7	8	9	2	11	**4**	13	14	1	16	17	18	19	20

Now that we know `K[3]`, the next step is to find `K[4]`. Here, we search the traffic for IVs of the form ⟨4,-1,X⟩, but the general strategy above still works.

The *[FMS01]* attack, as originally described, requires on average about 60 weak IVs for each secret key byte, and a weak IV turns up about every 2^{16} packets. Each key byte, however, requires a different IV, so any one IV has one chance per key byte to be a weak IV. To guess a key thus requires $60 \times 65536 \simeq 4$ million packets.

But we can do quite a bit better. In *[TWP07]*, the number of packets needed is only about 40,000 on average. At that point the attack is entirely practical.

This vulnerability was used in the attack on TJX Corp mentioned in the opening section.

22.8 Diffie-Hellman-Merkle Exchange

How do two parties agree on a secret key K? They can meet face-to-face, but that can be expensive or even not possible. We would like to be able to encrypt, for example, online shopping transactions, which often occur at the spur of the moment.

Diffie, Hellman and Merkle

Diffie and Hellman's paper *[DH76]* appeared in 1976, but they reference Merkle's later-published paper *[RM78]* as "submitted". In *[MH04]*, Hellman suggested that the name of the key-exchange algorithm here should bear the names of all three contributors. The algorithm was apparently discovered in 1974 by Malcolm Williamson at GCHQ, but Williamson's work was classified.

One approach is to use public-key encryption, below, but that came two years later. The Diffie-Hellman-Merkle approach, from *[DH76]* and *[RM78]*, allows private key exchange without public keys.

The goal here is for Alice and Bob to exchange information over a public network so that, at the end, Alice and Bob have determined a common key, and no eavesdropper can determine it without extraordinary computational effort.

As a warmup, we begin with a simple example. Alice and Bob each choose a number; say Alice chooses 7 and Bob chooses 9. Alice sends to Bob the value $A = 3^7 = 2187$, and Bob sends to Alice the value $B = 3^9 = 19683$. Alice then computes $B^7 = 3^{9\times7}$, and Bob computes $A^7 = 3^{7\times9}$. If they each use 32-bit arithmetic, they have each arrived at 2111105451 (the exact value is 1144561273430837494885949696427).

The catch here is that any intruder with a calculator can recover 7 from A and 9 from B, by factoring. But if we make the arithmetic a little more complicated, this becomes extremely difficult.

In the actual protocol, Alice and Bob agree, publicly, on a large prime p (typically several hundred digits) and a small value g. Alice then chooses $a<p$, and sends to Bob the value $A = g^a \bmod p$. Similarly, Bob chooses $b<p$ and sends to Alice the value $B = g^b \bmod p$.

As before, Alice and Bob now compute $B^a \bmod p$ and $A^b \bmod p$, which both equal $g^{a\times b} \bmod p$. This becomes their shared key.

The question is whether an eavesdropper who knows A and B and g can compute a and b. This is the **discrete logarithm problem**, as in some mod-p sense $a = \log_g(A)$. It is indeed believed to be computationally intractable, provided that g is a so-called primitive root modulo p (meaning that the values $g^1 \bmod p$, $g^2 \bmod p$, ..., $g^{p-1} \bmod p$ are all distinct). A naive attempt to compute a from A is to try each g^i for $i<p$ until one finds a match; this is much too slow to be practical.

There is some work involved in finding a suitable primitive root g given p, but that can be public, and is usually not *too* time-consuming. However, p and g should be updated at regular intervals, and the same p (and g) should not be shared with other sites. Otherwise one may be vulnerable to the **logjam** attack described in *[ABDGHSTVWZ15]*. For a fixed prime p, it turns out that relatively fast discrete-logarithm computation is possible if an attacker is able to pre-compute a very large table. For 512-bit primes, the average logarithm-calculation time in *[ABDGHSTVWZ15]* was 90 seconds, once the table was constructed. Time for calculation of the table itself took almost 10^5 core-hours, though that could be compressed into as little as one week with highly parallel hardware. By comparison, the authors were able to factor 512-bit RSA keys in about 5,000 core-hours; see *22.9.1.2 Factoring RSA Keys*.

The authors of *[ABDGHSTVWZ15]* conjecture that the NSA has built such a table for some well-known (and commonly used) 1024-bit primes. However, for those with fewer computational resources, the logjam attack also includes strategies for forcing common server implementations to downgrade to the use of 512-bit primes using TLS implementation vulnerabilities; *cf* the POODLE attack (first sidebar at *22.10 SSH and TLS*).

The Diffie-Hellman-Merkle exchange is vulnerable to a **man-in-the-middle attack**, discussed in greater detail in *22.9.3 Trust and the Man in the Middle*. If Mallory intercepts Alice's g^a and sends Bob his own $g^{a'}$, and similarly intercepts Bob's g^b and replaces it with his own $g^{b'}$, then the Alice–Mallory link will negotiate key $g^{a\times b'}$ and the Mallory-Bob link will negotiate $g^{a'\times b}$. Mallory then remains in the middle, intercepting, decryping, re-encrypting and forwarding each message between Alice and Bob. The usual fix is for Alice and Bob to *sign* their g^a and g^b; see *22.9.2 Forward Secrecy*.

22.8.1 Fast Arithmetic

It is important to note that the other arithmetic operations here – *eg* calculating $g^a \bmod p$ – can be done in polynomial time with respect to the number of binary digits in p, sometimes called the **bit-length** of p. This is not completely obvious, as the naive approach of multipying out $g\times g\times\ldots\times g$, a times, takes O(p) steps when $a\simeq p$, which is exponential with respect to the bit-length of p. The trick is to use repeated squaring: to compute g^{41}, we note 41 = 101001 in binary, that is, $41 = 2^5 + 2^3 + 1$, and so

$$g^{41} = ((((g^2)^2)^2)^2)^2 \times ((g^2)^2)^2 \times g$$

A simple python3 implementation of this appears at *22.13 RSA Key Examples*.

It is perhaps also worth noting that even *finding* large primes is not obviously polynomial in the number of digits. We can, however, use fast probabilistic primality testing, and note that the Prime Number Theorem guarantees that the number of candidates we must test to find a prime near a given number n is O(log n).

22.8.2 Simultaneous Authentication of Equals

The Simultaneous Authentication of Equals protocol, or **SAE**, is an approach to *mutual* password authentication: each side has the password, and each side verifies to the other that it knows the password. Note that an exchange of passwords cannot achieve this. It can be compared to the WPA2 four-way handshake used in Wi-Fi, *3.7.5.1 WPA2 Four-way handshake*; in fact, in the WPA3 standard SAE is partly replacing the four-way handshake. It is described in *[DH08]*; a close variant known as Dragonfly is described in **RFC 7664**.

A side effect of SAE is that each side generates a shared secret that can be used as an encryption key. In this sense, SAE acts as a key-exchange protocol; *cf 22.8 Diffie-Hellman-Merkle Exchange*.

The SAE protocol will be presented here using arithmetic modulo a large prime number, although elliptic curves (sidebar at *22.9.1 RSA*) can also be used.

Each side, Alice and Bob, initially knows a password, likely a string, and each side agrees on a large prime number p. Each side also agrees on a number g (a generator) such that the *order* of g modulo p is a large prime q. Specifically, this means g^q mod p = 1, but g^n mod p $\neq$ 1 for any 1<n<q.

Each side determines, from the password, a numeric value PW < p. This starts with a hash of the password, modulo p, to create a seed value. PW is then calculated as $\text{seed}^{(p-1)/q}$ mod p. This ensures PW^q mod p = 1.

The authentication starts with each side choosing random values rand and mask, each less than the prime q. Alice's values are randA and maskA, and likewise for Bob. Each side then computes the following:

$$\text{scal} = (\text{rand} + \text{mask}) \bmod q$$
$$\text{elem} = PW^{-\text{mask}} \bmod p$$

The names here reflect the fact that scal is a scalar value, and elem is an element of the group generated by g. The negative exponent in the second formula here warrants some explanation; $PW^{-\text{mask}}$ represents the (unique modulo p) number X such that X × PW^{mask} mod p = 1.

Let Alice's scal and elem be denoted scalA and elemA, and Bob's scalB and elemB. The first step of the protocol is for Alice and Bob to exchange their scal and elem values.

Each side now computes a secret K as follows. Alice computes

$$K = (PW^{\text{scalB}} \times \text{elemB})^{\text{randA}} \bmod p$$

while Bob computes

$$K = (PW^{\text{scalA}} \times \text{elemA})^{\text{randB}} \bmod p$$

The two values of K are in fact equal, as we can see by expanding Alice's version. We replace scal with rand+max+Nq, the Nq reflecting the fact that we reduced mod q. However, PW^q mod p = 1, so this term drops out.

$$\begin{aligned} K &= (PW^{\text{scalB}} \times \text{elemB})^{\text{randA}} \\ &= (PW^{(\text{randB}+\text{maskB}+Nq)} \times PW^{-\text{maskB}})^{\text{randA}} \end{aligned}$$

$= (\mathrm{PW}^{\mathrm{randB}})^{\mathrm{randA}}$

$= \mathrm{PW}^{\mathrm{randB} \times \mathrm{randA}}$

This last value is the same for both Alice and Bob.

The second, and final, step of the protocol involves an exchange of verification tokens. First, the actual key k is computed from K. Alice then sends to Bob a value like the following, where H() is a suitable secure-hash function.

Hash(k⌒elemA⌒scalA⌒elemB⌒scalB)

Bob sends something similar. Alice and Bob have now each verified to the other that they knew the password.

If Alice did not really know PW, and used an incorrect value instead, then she will get a different K, and Bob will know when the second-step tokens are exchanged.

Perhaps more importantly, an attacker who eavesdrops on the Alice-Bob exchange obtains nothing of use; there is no known offline dictionary attack. In this, SAE is much more resistant to eavesdropping than, say, the WPA2 handshake used by Wi-Fi. The second SAE step is protected by a secure-hash function, and, for the first step, knowing scal and elem convey nothing about PW, in large part due to the difficulty of solving the discrete logarithm problem.

The SAE protocol does require that Alice and Bob each keep a copy of the password cleartext, not a copy of the password hash. This seems unavoidable, given that Bob and Alice must each authenticate to one another.

22.9 Public-Key Encryption

In public-key encryption, each party has a public encryption key K_E and a private decryption key K_D. Alice can publish her K_E to the world. If Bob has Alice's K_E, he can encrypt a message with it and send it to her; only someone in possession of Alice's K_D can read it. The fundamental idea behind public-key encryption is that knowledge of K_E should not offer any meaningful information about K_D.

22.9.1 RSA

Public-key encryption was outlined in *[DH76]*, but the best-known technique is arguably the RSA algorithm of *[RSA78]*. (The RSA algorithm was discovered in 1973 by Clifford Cocks at GCHQ, but was classified.)

To construct RSA keys, one first finds two very large primes p and q, perhaps 1024 binary digits each. These primes should be chosen at random, by choosing random N's in the right range and then testing them for primality until success. Let $n = p \times q$.

It now follows from Fermat's little theorem that, for any integer m, $m^{(p-1)(q-1)} = 1 \bmod n$ (it suffices to show $m^{(p-1)(q-1)} = 1 \bmod p$ and $m^{(p-1)(q-1)} = 1 \bmod q$).

One then finds positive integers e and d so that $e \times d = 1 \bmod (p-1)(q-1)$; given any e relatively prime to both p-1 and q-1 it is possible to find d using the Extended Euclidean Algorithm (a simple Python implementation appears below in *22.13 RSA Key Examples*). From the claim in the previous paragraph, we now know $m^{e \times d} = (m^e)^d = m \bmod p$.

If we take m as a message (that is, as a bit-string of length less than the bit-length of n, rather than as an integer), we encrypt it as $c = m^e \bmod n$. We can decrypt c and recover m by calculating $c^d \bmod n = m^{e\times d} \bmod n = m$.

The public key is the pair $\langle n,e\rangle$; the private key is d.

Elliptic-curve cryptography

One of the concerns with RSA is that a faster factoring algorithm will someday make the encryption useless. A newer alternative is the use of *elliptic curves*; for example, the set of solutions modulo a large prime p of the equation $y^2 = x^3 + ax + b$. This set has a natural (but nonobvious) product operation completely unrelated to modulo-p multiplication, and, as with modulo-p arithmetic, finding n given $B = A^n$ appears to be quite difficult. Several cryptographic protocols based on elliptic curves have been proposed; see Wikipedia.

As with Diffie-Hellman-Merkle (*22.8.1 Fast Arithmetic*), the operations above can all be done in polynomial time with respect to the bit-lengths of p and q.

The theory behind the security of RSA is that, if one knows the public key, the only way to find d in practice is to factor n. And factoring is a hard problem, with a long history. There are much faster ways to factor than to try every candidate less than $\sqrt{n}$, but it is still believed to require, in general, close-to-exponential time with respect to the bit-length of n. See *22.9.1.2 Factoring RSA Keys* below.

RSA encryption is usually thousands of times slower than comparably secure shared-key ciphers. However, RSA needs only to be used to encrypt a secret key for a shared-key cipher; it is this latter key that actually protects the document.

For this reason, if a message is encrypted for multiple recipients, it is usually not much larger than if it were encrypted for only one: the additional space needed for each additional recipient is just the encrypted key for the shared-key cipher.

Similarly, for digital-signature purposes Alice doesn't have to encrypt the entire message with her private key; it is sufficient (and much faster) to encrypt a secure hash of the message.

If Alice and Bob want to negotiate a session key using public-key encryption, it is sufficient for Alice to know Bob's public key; Alice does not even need a public key. Alice can choose a session key, encrypt it with Bob's public key, and send it to Bob.

We will look at an actual calculation of an RSA key and then an encrypted message in *22.13 RSA Key Examples*.

22.9.1.1 RSA and Digital Signatures

RSA can also be used for strong digital signatures (as can any public-key system in which the roles of the encryption and decryption keys are symmetric, and can be reversed). Suppose, as above, Alice's public key is the pair $\langle n,e\rangle$ and her private key is the exponent d. If Alice wants to sign a message m to Bob, she simply encrypts it using the exponent d from her *private* key; that is, she computes $c = m^d \bmod n$. Bob, along with everyone else in the world, can then decrypt the resulting ciphertext c with Alice's *public* exponent e (that is, c^e). If this yields a valid message, then Alice (or someone in possession of her key) must have been the sender.

Generally, for signature purposes Alice will encrypt not her entire message but simply a secure hash of it, and then send both the message and the RSA-encrypted hash. Bob decrypts the encrypted hash, and the signature checks out if the result matches the hash of the corresponding message.

The advantage of a public-key signature over an HMAC signature is that the former uses a key with long-term persistence, and can be used to identify the *individual* who sent the message. The drawback is that HMAC signatures are much faster.

It is common for Alice to sign her message to Bob and then encrypt the message plus signature – perhaps with Bob's public key – before sending it all to Bob.

Just as Alice probably prefers not to sign undated documents with her handwritten signature, she will likely prefer to apply her digital signature only to documents containing a timestamp. This helps deter replay attacks.

22.9.1.2 Factoring RSA Keys

The security of RSA depends largely on the difficulty in factoring the key modulus n = pq (though it is theoretically possible that an RSA vulnerability exists that does not entail factoring). As of 2015, the factoring algorithm that appears to be the fastest for larger keys is the so-called number-field sieve; an early version of this technique was introduced in *[JP88]*. A heuristic estimate of the time needed to factor a number n via this algorithm is $\exp(1.923\times\log(n)^{1/3}\times\log(\log(n))^{2/3})$, where $\exp(x) = e^x$ and log(x) is the natural logarithm. To a first approximation this is $\exp(k\times L^{1/3})$, where L is the bit-length of N. This is sub-exponential (because of the exponent 1/3) but still extremely slow.

Side Channels

Those placing serious reliance on the relative security levels outlined here should also be aware of the existence of so-called side-channel attacks, *eg* making use of electromagnetic leakage from a computer to infer a key.

The first factorization of an RSA modulus with 512 bits occurred in 1999 and was published in *[CDLMR00]*; this was part of the RSA factoring challenge. It took seven calendar months, using up to 300 processors in parallel. Fifteen years later, *[ABDGHSTVWZ15]* reported being able to do such a factorization in eight days on a 24-core machine (~5,000 core-hours), or in seven hours using 1800 cores. Some of the speedup is due to software implementation improvements, but most is due to faster hardware.

If we normalize the number-field-sieve factoring time of a 512-bit n to 1, we get the following table of estimated relative costs for factoring larger n:

key length in bits	relative factoring time
512	1
1024	8,000,000
1536	8×10^{11}
2048	8×10^{15}
3072	3×10^{22}
4096	8×10^{27}

From this we might conclude that 1024-bit keys are potentially breakable by a *very* determined and well-funded adversary, while the 2048-bit key length appears to be much safer. In 2011 NIST's recommended key length for RSA encryption increased to 2048 bits (publication 800-131A, Table 6). This is quite a bit larger than the 128-to-256-bit recommendation for shared-key ciphers, but RSA factoring attacks are *much* faster than trying all keys by brute force.

Note that some parts of the factoring algorithm are highly parallelizable while other parts are less so; relative factoring times when using highly parallel hardware may therefore differ quite a bit. See also *22.13.1 Breaking the key*.

22.9.2 Forward Secrecy

Suppose Alice and Bob exchange a shared-key-cipher session key K_S using their RSA keys. Later, their RSA keys are compromised. If the attacker has retained Alice and Bob's prior communications, the attacker can go back and decrypt K_S, and then use K_S to decrypt the entire session protected by K_S.

This is not true, however, if Alice and Bob had used Diffie-Hellman-Merkle key exchange. In that case, there is no encryption used in the process of negotiating K_S, so no later encryption compromise can reveal K_S.

This property is called **forward secrecy**, or, sometimes, *perfect* forward secrecy (other times, perfect forward secrecy adds the further requirement that the compromise of any other session key negotiated by Alice and Bob does not reveal information about K_S).

The advantage of public-key encryption, however, is that Alice can sign the key K_S she sends to Bob. Assuming Bob is confident he has Alice's real public key, a man-in-the-middle attack (*22.9.3 Trust and the Man in the Middle*) becomes impossible.

It is now common for public-key encryption to be used to sign all the transactions that are part of the Diffie-Hellman-Merkle exchange. When this is done, Alice and Bob gain both forward secrecy *and* protection from man-in-the-middle attacks.

22.9.3 Trust and the Man in the Middle

Suppose Alice wants to send Bob a message. Where does she find Bob's public key E_B?

If Alice goes to a public directory that is not completely secure and trustworthy, she may find a key that in fact belongs to Mallory instead. Alice may now fall victim to a **man-in-the-middle** attack, like that at the end of *22.8 Diffie-Hellman-Merkle Exchange*:

- Alice encrypts Bob's message using Mallory's public key E_{Mal}, thinking it is Bob's
- Mallory intercepts and decrypts the message, using his own decryption key D_{Mal}
- Mallory re-encrypts the message with Bob's real public key E_{Bob}
- Bob decrypts the message with D_{Bob}

Despite Mallory's inability to break RSA directly, he has read (and may even modify) Alice's message.

Alice can, of course, get Bob's key directly from Bob. Of course, if Alice met Bob, the two could also exchange a key for a shared-key cipher.

Wi-Fi in the Middle

One of the easiest settings at which to launch a man-in-the-middle attack is a public Wi-Fi hotspot, either as the hotspot owner or by setting up a rogue access point to which some customers mistakenly connect.

We now come to the mysterious world of trust. Alice might trust Charlie for Bob's key, but not Dave. Alice doesn't have to meet Charlie to get Bob's key; all she needs is

- a trusted copy of Charlie's public key
- a copy of Bob's public key, together with Bob's name, *signed by* Charlie

At the same time, Alice might trust Dave but not Charlie for Evan's key. And Bob might not trust Charlie *or* Dave for Alice's key. Mathematically, the trust relationship is neither symmetric nor transitive. To handle the possibility that trust might erode with time, signed keys often have an expiration date.

At the small scale, **key-signing parties** are sometimes held in which participants exchange some keys directly and others indirectly through signing. This approach is sometimes known as the **web of trust**. At the large scale, **certificate authorities** (*22.10.2.1 Certificate Authorities*) are entities built into the TLS framework (*22.10.2 TLS*) that verify that a website's public key is as claimed; you are implicitly trusting these certificate authorities if your browser vendor trusts them. Both the web of trust and certificate authorities are examples of "public-key infrastructure" or **PKI**, which is, broadly, any mechanism for reliably tying public keys to their owners. For applications of public-key encryption that manage to avoid the need for PKI, by use of **cryptographically generated addresses**, see *8.6.4 Security and Neighbor Discovery* and the discussion of .onion addresses at the end of *7.8 DNS*.

22.9.4 End-to-End Encryption

Many communications applications are based on the model of encryption from each end-user to the central server. For example, Alice and Bob might both use https (based on TLS, *22.10.2 TLS*) to encrypt their interactions with their email provider. This means Alice and Bob are now trusting that provider, who decrypts messages from Alice, stores them, and re-encrypts them when delivering them to Bob.

This model does protect Alice and Bob from Internet eavesdroppers who have not breached the security of the email provider. However, it also allows government authorities to order the email provider to turn over Alice and Bob's correspondence.

If Alice and Bob do not wish to trust an intermediary, or their (or someone else's) government, they need to implement **end-to-end** encryption. That is, Alice and Bob must negotiate a key, use that key to encrypt messages between them, and not divulge the key to anyone else. This is quite a bit more work for Alice and Bob, and even more complicated if Alice wishes to use end-to-end encryption with a large number of correspondents.

Of course, even with end-to-end encryption Alice may still be compelled by subpoena to turn over her correspondence with Bob, but that is a different matter. Alice's private key may also be seized under a search warrant. It is common (though not universal) to protect private keys with a password; this is good practice, but protecting a key having an effective length of 256 bits with a password having an effective length of 32 bits leaves something to be desired. The mechanisms of *22.6.2 Password Hashes* provide only limited relief. The mechanism of *22.9.2 Forward Secrecy* may be more useful here, assuming Alice can communicate to Bob that her previous key is now compromised; see also *22.10.2.3 Certificate revocation*.

22.10 SSH and TLS

We now look at two encryption mechanisms in popular use on the Internet. The first is the Secure Shell, **SSH**, used to allow login to remote systems, remote command execution, file transfer and even some forms of VPN tunneling. Public-key-encryption is optional; if it is used, the public keys are generally transported manually.

POODLE

The 2014 POODLE attack was based in part on a long-known flaw in SSL 3.0. But SSL 3.0 is fifteen years obsolete (though it was not officially deprecated until 2015); the other flaw was a broken version-negotiation implementation in which the disruption of a few packets by an attacker could force both client and server to settle on SSL 3.0 even when both supported the latest TLS version. See *22.10.2.4 TLS Connection Setup*.

The second example is the Transport Layer Security protocol, **TLS**, which is a successor of the Secure Sockets Layer, or **SSL**. Many people still refer to TLS by the SSL name even though TLS replaced SSL in 1999, though, to be fair, TLS is effectively an update of SSL, and TLS v1.0 could easily have been named SSL v4.0. TLS is used to encrypt web traffic for the HTTP Secure protocol, **https**; it is also used to encrypt traffic for several other applications and *can* be used, with appropriate programming, for any application.

If Alice wants to contact a server S using either SSH or TLS, at some point she will have to trust a public key claimed to be from S. A common approach with SSH is for Alice to accept the key on faith the first time it is presented, but then to save it for all future verifications. Under TLS, the key from S that Alice receives will have been signed by a certificate authority (*22.10.2.1 Certificate Authorities*); Alice presumably trusts this certificate authority. (SSH does now support certificate authorities too, but their use with SSH seems not yet to be common.)

Both SSH and TLS eventually end up negotiating a shared-secret session key, which is then used for most of the actual data encryption.

22.10.1 SSH

The SSH protocol establishes an encrypted communications channel between two hosts, after establishing the identities of each endpoint. Its primary use is to enable secure remote-command execution with input and output via the secure channel; this includes the remote execution of an interactive shell, which is in effect a telnet-style terminal login with encryption. The companion program `scp` uses the SSH protocol to implement secure file transfer. Finally, ssh supports secure port forwarding from one machine (and port) to another; unrelated applications can then connect to one machine and find themselves securely talking to another. The current version of the SSH protocol is 2.0, and is defined in **RFC 4251**. The authentication and transport sub-protocols are defined in **RFC 4252** and **RFC 4253** respectively.

One of the first steps in an SSH connection is for the endpoints to negotiate which secret-key cipher (and mode) to use. Support of the following ciphers is "recommended" and there is a much longer list of "optional" ciphers (which include RC4 and Blowfish); the table below includes those added by **RFC 4344**:

cipher	modes	nominal keylength
3DES	CBC, CTR	168 bits
AES	CBC, CTR	192 bits
AES	CTR	128 bits
AES	CTR	256 bits

SSH supports a special name format for including new ciphers for local use.

The SSH protocol also allows the endpoints to negotiate a public-key-encryption mechanism, a secure-hash function, and even a key-exchange algorithm although only minor variants of Diffie-Hellman-Merkle key exchange are implemented.

If Alice wishes to connect to a server S, the server clearly wants to verify Alice's identity, either through a password or some other means. But it is just as important for Alice to verify the identity of S, to prevent man-in-the-middle attacks and the possibility that an attacker masquerading as S is simply collecting Alice's password for later use.

The SSH protocol is not designed to minimize the number of round-trip packet exchanges, making SSH connection setup quite a bit slower than TLS connection setup (*22.10.2.4 TLS Connection Setup*). A connection may involve quite a few round trips to get started: the TCP three-way handshake, the protocol version exchange, the "Key Exchange Init" exchange, the actual Diffie-Hellman-Merkle key exchange, the "NewKeys" exchange, and the "Service Request" exchange (all but the first are documented in **RFC 4253**). Still, multi-second delays can usually be reduced through performance tuning; enabling diagnostic output (*eg* with the `-v` option) is often helpful. A common delay culprit is a server-side DNS lookup of the client, fixable with a server-side configuration setting of `UseDNS no` or the equivalent.

In the following subsections we focus on the "common" SSH configuration and ignore some advanced options.

22.10.1.1 Server Authentication

To this end, one of the first steps in SSH connection negotiation is for the server to send the public half of its **host key** to Alice. Alice verifies this key, which is typically in her `known_hosts` file. Alice also asks S to sign something with its host key. If Alice can then decrypt this with the public host key of S, she is confident that she is talking to the real S.

If this is Alice's first attempt to connect to S, however, she should get a message like the one below:

> The authenticity of host 'S (10.2.5.1)' can't be established.
>
> RSA key fingerprint is da:2e:e3:94:84:6b:bf:6d:2f:e4:c3:76:68:72:a5:a0.
>
> Are you sure you want to continue connecting (yes/no)?

If Alice is cautious, she may contact the administrator of S and verify the key fingerprint. More likely, she will simply choose "yes", in which case the host key of S will be added to her own `known_hosts` file; this latter strategy is sometimes referred to as **trust on first use**.

If she later re-connects to S after the host key of S has been changed, she will get a rather more dire message, and, under the default configuration, she will not be allowed to continue until she manually removes the old, now-incorrect, host key for S from her `known_hosts` file.

See also the ARP-spoofing scenario at *7.9.2 ARP Security*, and the comment there about how this applies to SSH.

SSH v2.0 now also supports a certificate-authority mechanism for verifying server host keys, replacing the `known_hosts` file.

22.10.1.2 Key Exchange

The next step is for Alice's computer and S to negotiate a session key. After the cipher and key-exchange algorithms are negotiated, Alice's computer and S use the chosen key-exchange algorithm to agree on a session key for the chosen cipher.

The two directions of the connection generally get different session keys. They can even use different encryption algorithms.

At this point the channel is encrypted.

22.10.1.3 Client Authentication

The server now asks Alice to authenticate herself. If password authentication is used, Alice types in her password and it and her username are sent to the server, over the now-encrypted connection. This does expose the server to brute-force password-guessing attacks, and is not infrequently disabled.

Alice may also have set up **RSA authentication** (other types of public-key authentication are also possible). For this, Alice must create a public/private key pair (often in files `id_rsa.pub` and `id_rsa`), and the public key must have been previously installed on S. On Unix-based systems it is often installed on S in the `authorized_keys` file in the `.ssh` subdirectory of Alice's home directory. If Alice now sends a message to S signed by her private key, S is in a position to verify the signature. If this succeeds, Alice can log in without supplying a password to S.

It is common though not universal practice for Alice's private-key file (on her own computer) to be protected by a password. If this is the case, Alice will need to supply that password, but it is used only on Alice's end of the connection. (The default password hash is MD5, which may not be a good choice; see *22.6.2 Password Hashes*.)

Public-key authentication is tried before password authentication. If Alice has created a public key, it is likely to be tried even if she has not copied it to S.

If S is set up to *require* public-key authentication, Alice may not have any direct way to install her public key on S herself, and may need the cooperation of the administrators of S. It is possible that the latter will send Alice a public/private keypair chosen by them specifically for S, rather than allowing Alice to choose her own keypair. The standard SSH user configuration does support different private keys for different servers.

In several command-line implementations of ssh, the various stages of authentication can be observed from the client side by using the `ssh` or `slogin` command with the `-v` option.

22.10.1.4 The Session

Once an SSH connection has started, a new session key is periodically negotiated. **RFC 4253** recommends this after one hour or after 1 GB of data is exchanged.

Data is then sent in packets generally with size a multiple of the block size of the cipher. If not enough data is available, *eg* because only a single keystroke (byte) is being sent, the packet is padded with random data as needed. *Every* packet is required to be padded with at least four bytes of random data to thwart attacks based on known plaintext/ciphertext pairs. Included in the *encrypted* part of the packet is a byte indicating the length of the padding.

22.10.2 TLS

Transport Layer Security, or TLS, is an IETF extension of the Secure Socket Layer (SSL) protocol originally developed by Netscape Communications. SSL went through published versions 2.0 and 3.0; the latter was introduced in 1996 and was officially deprecated by **RFC 7568** in 2015 (see the POODLE sidebar above at *22.10 SSH and TLS*). TLS 1.0 was introduced in 1999, as the IETF took ownership of the specification from Netscape. The current version of TLS is 1.2, specified in **RFC 5246** (plus updates listed there). A draft of TLS 1.3 is in progress (2018); see draft-ietf-tls-tls13.

The original and still primary role for TLS is encrypting web connections and verifying for the client the authenticity of the server. TLS can, however, be embedded in any network application.

Unlike SSH, *client* authentication, while possible, is not common; web servers often have no pre-existing relationship with the client. Also unlike SSH, the public-key mechanisms are all based on established **certificate authorities**, or CAs, whereas the most common way for an SSH server's host key to end up on a client is for it to have been accepted by the user during the first connection. Browsers (and other TLS applications as necessary) have embedded lists of certificate authorities, known as **trust stores**, trusted by the browser vendor. SSH requires no such centralized trust.

If Bob wishes to use TLS on his web server S_{Bob}, he must first arrange for a certificate authority, say CA_1, to sign his **certificate**. A certificate contains the full DNS name of S_{Bob}, say bob.int, a public key K_S used by S_{Bob}, and also an expiration date. TLS uses the ITU-T X.509 certificate format.

The `.int` domain

For Bob to actually have a domain name in the `.int` top-level domain, as in the example here, Bob's organization must be established by international treaty.

Now imagine that Alice connect to Bob's server S_{Bob} using TLS. Early in the process S_{Bob} will send her its signed certificate, claiming public key K_S. Alice's browser will note that the certificate is signed by CA_1, and will look up CA_1 on its list of trusted certificate authorities. If found, the next step is for the browser to use CA_1's public key, also on the list, to verify the signature on the certificate S_{Bob} sent.

If everything checks out, Alice's browser now knows that CA_1 certifies bob.int has public key K_S. As S_{Bob} has presented this key K_S, and is able to verify that it possesses the matching private key, this is proof that S_{Bob} is legitimately the server with domain name bob.int.

Assuming, of course, that CA_1 is correct.

As with SSH, once Alice has accepted the public key K_S of S_{Bob}, a secret-key cipher is negotiated and the remainder of the exchange is encrypted.

22.10.2.1 Certificate Authorities

A **certificate authority**, or CA, is just an entity in the business of signing certificates: messages that include a name S_{Bob} and a verified public key K_S. The purpose of the certificate authority is to prevent man-in-the-middle attacks (*22.9.3 Trust and the Man in the Middle*); Alice wants to be sure she is not really connected to S_{Bad} instead of to S_{Bob}.

What the CA is actually signing is that the particular public key K_S belongs to domain bob.int, just like Charlie might sign Bob's public key on an individual basis. The difference here is that the CA is likely to be a large organization, and the CA's public key is likely to be embedded in the network application software somewhere. (Real CA's usually have a two-layer key signing arrangement, in which Bob's public key would be signed by one of the CA's subsidiary keys, but the effect is the same.)

A certificate specific for bob.int will not work for **www**.bob.int, even though both DNS names might direct to the same server S_{Bob}. If S_{Bob} presents a certificate for bob.int to a browser that has arrived at S_{Bob} via the url http://www.bob.int, the user should see a warning. It is straightforward, however, for the server either to support two certificates, or (if the certificate authority supports this) a single certificate for www.bob.int with a Subject Alternative Name also entered on the certificate for bob.int alone. It is also possible to obtain "wildcarded" certificates for *.bob.int (though this does not match bob.int; it also does not match names with two additional levels such as www.foo.bob.int). **RFC 6125**, §7.2, recommends against wildcard certificates, though they remain widely used.

Firefox Certificates

As of this writing, certificate authorities for the popular Firefox browser are found in Preferences → Advanced → View Certificates. It is an interesting list, if only because, for such highly trusted organizations, few people have heard of more than a handful of them. The policy for becoming a Firefox CA is here.

Popular browsers all use preset lists of CAs provided by (and presumably trusted by) either the browser vendor or the host operating-system vendor. **If** Alice is also willing to trust these CAs, she can feel comfortable using the key she receives to send private messages to Bob. That "if", however, is sometimes controversial. Just because Alice trusts her browser and operating system (*eg* not to contain malware), that does *not* automatically imply that Alice should trust these vendors' judgment when it comes to CAs.

On the face of it, Bob's certificate authority CA_1 is just signing that domain name bob.int has public key K_S. This isn't *quite* the same, however, as attesting that the certificate-signing request actually came from Bob. All depends on how thorough CA_1 is in checking the identity of its customer, and since those customers typically choose the least expensive CA, there is sometimes an incentive to cut corners.

A certificate authority's failure to verify the identity of the party making the certificate-signing request can enable a man-in-the-middle attack (*22.9.3 Trust and the Man in the Middle*). Suppose, for example, that Mallory is able to obtain a signed certificate from another certificate authority CA_2 linking bob.int to key K_{bad} controlled by Mallory. If Mallory can now position his server S_{Bad} in the middle between Alice and S_{Bob},

Alice —— S_{Bad} —— S_{Bob}

then Alice can be tricked into connecting to S_{Bad} instead of S_{Bob}. Alice will request a certificate, but from S_{Bad} instead of S_{Bob}, and get Mallory's from CA_2 instead of Bob's actual certificate from CA_1. Mallory opens a second connection from S_{Bad} to S_{Bob} (this is easy, as Bob makes no attempt to verify Alice's identity), and forwards information from one connection to the other. As far as TLS is concerned everything checks out. From Alice's perspective, Mallory's false certificate vouches for the key K_{bad} of S_{Bad}, CA_2 has signed this certificate, and CA_2 is trusted by Alice's browser. There is no "search" at any point through the other CAs to see if any of them have any contrary information about S_{Bob}. In fact, there is not necessarily even contact with CA_2, though see *22.10.2.3 Certificate revocation* below.

Certificate Errors

Sometimes certificate signatures fail to verify, either because they are expired or were improperly signed or the certificate authority is not recognized. The browser then receives the appropriate TLS error message and communicates the problem to the user. Examples can be found at badssl.com; click the buttons there to see different errors.

If the certificate authority CA_1 were also the domain registrar with whom Bob registered the DNS name bob.int, it would be especially well-positioned to verify that Bob is really the owner of the bob.int domain. But this is not generally the case. Quite often, the CA's primary verification method is to send an email to, say, bob@bob.int (or perhaps to the DNS administrative contact listed in the WHOIS database for domain bob.int). If someone responds to this email, it is assumed they must be legitimately part of the bob.int domain. Another authentication strategy is for CA_1 to send a special file to be placed at, say, bob.int/foo/bar/special.html; again, the idea is that only the domain owner would be able to do this. Note, however, that in the man-in-the-middle attack above, it does not matter what CA_1 does; what matters is the verification policy used by CA_2.

If Alice is very careful, she may click on the "lock" icon in her browser and see that the certificate authority signing her connection to S_{Bad} is CA_2 rather than CA_1. But if Alice has a secure way of finding Bob's "real" certificate authority, she might as well use it to find Bob's key K_S. As she often does not, this is of limited utility in practice.

The second certificate authority CA_2 might be a legitimate certificate authority that has been tricked, coerced or bribed into signing Mallory's certificate. Alternatively, it might be Mallory's own creation, inserted by Mallory into Alice's browser through some other vulnerability.

Superfish

The "Superfish" vulnerability of 2015 involved an operating-system module (based on LSP, WFP or, in principle, iptables) that could intercept all browser TLS connections (thus acting as the man in the middle), together with software that could act as a certificate authority, generating new certificates (like Mallory's from CA_2) on the fly. The apparent goal was to inject advertising into TLS-secured connections.

Mallory's machinations here do require both the man-in-the-middle attack and the bad certificate. If Alice is able to establish a direct connection with S_{Bob}, then the latter will send its true key K_S signed by CA_1.

As another attack, Mallory might obtain a certificate for b0b.int and hope Alice doesn't notice the spelling difference between B0B and BOB. When this is done, Mallory often also sends Alice a disguised link to b0b.int in the hope she will click on it. Unicode domain names make this even easier, as Unicode provides many character pairs that are different but which look identical.

Certificates tend to come with a number of constraints. First among them is that most certificates issued by CAs to end users (either corporations or individuals) are marked as "not a certificate authority", meaning that the certificate cannot be used to sign additional certificates. That is, the certificate trust relationship is *not transitive*. If Bob gets a certificate for bob.int, he cannot use it to sign a certificate for Alice or, for that matter, Google. The second major restriction is on names: a certificate is almost always tied to a specific DNS name. As described above, multiple names can be accepted via the "subject alternative name" attribute, or by using a "wildcard" name like `*.foo.com`. Finally, certificates often contain restrictions on how they can be used, in the "extended key usage" attribute. A typical entry is "TLS web server authentication", expressed as an OID (*21.3 SNMP Naming and OIDs*). These restrictions are enforced by the TLS library; it is possible outside that library to use the keys inside certificates for whatever you want.

Extended-Validation certificates were introduced in 2007 as a way of providing greater assurances that the certificate issued to bob.int was in fact generated by a request from Bob himself; Mallory should in theory have a much harder time obtaining an EV certificate for bob.int from CA_2. Desktop browsers that have secured a TLS connection using an EV certificate typically add the name of the domain owner, highlighted in green and/or with a green padlock icon, to the address bar. Financial institutions often use EV certificates. So does mozila.org.

Of course, if Alice does not know Bob is using an EV certificate, she can still be tricked by Mallory as above. Alternatively, perhaps Alice is using a mobile device; most mobile browsers give little if any visual indication of EV certificates. Because the majority of browsing today (2018) is done on mobile devices, this severely limits the usefulness of these certificates.

22.10.2.2 Certificate pinning

Another strategy intended as a more direct prevention of man-in-the-middle attacks is **certificate pinning**. Conceptually, Alice (or her browser) makes a note the first time she connects to S_{Bob} that the certificate authority is CA_1 or that Bob's public key is K_S, or both. Future connections to S_{Bob} must match at least one of these credentials. This form of pinning depends on Alice's having reached the real S_{Bob} on the first connection, sometimes called "trust on first use". A similar trust-on-first-use strategy is often (though not always) used with SSH, *22.10.1.1 Server Authentication*.

In the pinning protocol described in **RFC 7469**, it is S_{Bob} that initiates the pinning by including a "pin directive" in its initial HTTP connection. This requests Alice's browser to pin the desired certificates, though the pinned correspondence between a site and its keys is always maintained at the browser side. The pin directive also specifies which keys (K_S or CA_1 or both) are pinned, and for how long.

If S_{Bob} sends a pin directive for K_S, then Alice's browser remembers K_S, and any new certificate at S_{Bob} will be a mismatch. If S_{Bob} pins CA_1, then S_{Bob} can have CA_1 issue a new key and it will still pass pin-validation. If CA_1 is later compromised, though, S_{Bob} is not protected against any new rogue certificates it issues. (This is generally a minor concern, as CA_1 compromise will always mean that *new* visitors to S_{Bob} will be vulnerable to man-in-the-middle attacks.)

Bob can also obtain, along with K_S, a backup certificate $K_{S\text{-backup}}$, and have S_{Bob}'s pin directives pin both of these. Then, if K_S is compromised, $K_{S\text{-backup}}$ is ready to go. Hopefully, key compromise is a rare event, so there is a very small chance that both K_S and $K_{S\text{-backup}}$ are compromised within the pin's lifetime. Key compromise pretty much follows from any server compromise, however, and if intruders break in, $K_{S\text{-backup}}$ cannot be put into production until Bob is sure the site is secure. That may take some time.

If K_S and $K_{S\text{-backup}}$ *are* both compromised, and CA_1 wasn't pinned, S_{Bob} may simply become inaccessible to returning visitors. Automatic unpinning of revoked certificates would help, but certificate revocation (see following section) has its own difficulties. The user may have an option to disable pin validation for this particular site, but it's not supposed to be as simple as clicking "ok", and in any event if the site is your bank you may be loath to do this.

22.10.2.3 Certificate revocation

Suppose the key r underlying the certificate for bob.int has been compromised, and Mallory has the private key. Then, if Mallory can trick Alice into connecting to S_{Bad} instead of S_{Bob}, the original CA_1 will validate the connection. S_{Bad} can prove to Alice that it has the secret key corresponding to K_S, and all the certificate does is to attest that bob.int has key K_S. Mallory's deception works even if Bob noticed the compromise and updated S_{Bob}'s key to K_2; the point is that Mallory still has the original key, and CA_1's certificate attesting to that original key is still valid.

There is a mechanism by which a certificate can be revoked. Revocation information, however, must be kept at some central directory; a server can continue to serve up a revoked certificate and unless the clients actively check, they will be none the wiser. This is the reason certificates have **expiration dates**.

The original revocation mechanism was the global **certificate revocation list**. A newer alternative is the Online Certificate Status Protocol, **OCSP**, described in **RFC 6960**. If Alice receives a certificate signed by CA_1, she can send the serial number of the certificate to a designated "OCSP responder" run by or on behalf of CA_1. If the certificate is still valid, the responder site will return a signed confirmation of that.

Of course, an eavesdropper watching Alice's traffic arriving at the OCSP responder – and the OCSP responder itself – now knows that Alice is visiting bob.int. An eavedropper closer to Alice, however, knows that anyway.

More seriously, someone running a man-in-the-middle attack close to Alice can probably intercept and block Alice's OCSP request. If Alice receives no response, what should she do? Maybe there is a problem, but maybe the responder site is simply down or the Internet is simply slow. Most browsers that actually do revocation checks assume the latter – known as **soft fail** – making revocation checks of dubious value. The alternative of refusing to allow access to the original site – **hard fail** – leads to many false positives.

As of 2016, there is no generally accepted solution. One minimal approach, in the event of OCSP soft fail, is to use hard fail with EV certificates, or at least to downgrade EV certificates to ordinary ones upon OCSP non-response. Another approach is **OCSP stapling**, in which the server site periodically (perhaps daily) requests a signed and dated update from its CA's OCSP server indicating that the site's certificate is still valid. This OCSP report is then "stapled" to the `ServerHello` message (below), if requested by the client. This allows the client to verify that the certificate was still valid quite recently.

Of course, if the client *is* the victim of a man-in-the-middle attack then the (fake) server will not staple an OCSP validity report, and the client must fall back to the regular OCSP lookup process. But this case can be expected to be infrequent, making a hard fail after OCSP non-response a reasonable option.

Google's Chrome browser implements **CRLSets** in lieu of checking OCSP servers, described here. This is a list of revoked certificates downloaded regularly to the browser. Unfortunately, the full list is much too large, so CRLSets are limited to emergency revocations and certificate revocations due to key compromise; even then the list is not complete.

22.10.2.4 TLS Connection Setup

The typical TLS client-side user is interested in viewing a web page as quickly as possible, placing a premium on rapid negotiation of the TLS connection. If sites were to load noticeably more slowly when encryption was used, encryption might not be used routinely. As a result, the connection-setup process, known as the TLS **handshake protocol**, is designed to complete in two RTTs. The goals of the handshake protocol are to agree on the encryption and authentication mechanisms to be used, to provide authentication as necessary, and to negotiate the encryption keys. Here is an outline of a typical exchange, with some options omitted:

client (Alice)	server (Bob)
`ClientHello` • preferred TLS version • session ID (empty for new connections) • list of ciphers acceptable to client • ServerName (optional extension) • start of key exchange (optional)	
	`ServerHello` • chosen version • chosen cipher • server key exchange (optional) • server certificate (separate message)
(client now knows the key) optional client certificate optional other data (may be encrypted) `Finished` message	
	`Finished` message (encrypted)

The client initiates the connection by sending its `ClientHello` message, which contains its preferred TLS/SSL-protocol version number, a session identifier, some random bytes (a cryptographic nonce), and a list of cipher *suites*: tuples consisting of a key-exchange algorithm, a shared-key encryption algorithm and a secure-hash algorithm. The list of cipher suites is ranked by the client according to preference.

Many large servers host multiple websites through "virtual hosting". Each website will have its own certificate, and the server needs to be able to know which one to send in its reply. In settings like this, the client includes the server's DNS name in a ServerName extension; the Server Name Indication (**SNI**) is defined in **RFC 6066**.

Choosing your cipher

In theory, any TLS client can choose which cipher suites it will accept. In practice, for most software applications this is non-trivial. In the Firefox browser, see `about:config`, under security.ssl3 (yes, ssl3 is in theory not the same as tls).

The server then responds with its `ServerHello`. This contains the final protocol-version number, generally the maximum of the version proposed by the client and the highest version supported by the server. The `ServerHello` also contains the server's choice of a cipher suite from the client's list, and some more random bytes (the server's nonce). The server also sends its certificate, if requested (which it almost always is). Exactly which certificate is sent may depend on the server name sent by the client in the optional Server-Name extension. The certificate is considered to be a separate "message" at the TLS protocol level, not part of the `ServerHello`, but everything is sent together.

Having received the server's certificate, the client application's next step is to validate this certificate, by checking its signature against the client's list of trusted certificate authorities, the client's "trust store". This step does not involve any network communication. While most operating systems maintain a list of generally trusted certificates, the client application can trust all, some or none of this list; the client can also load application-specific certificates. If the certificate does *not* check out, the client is free to continue the connection, perhaps pausing to add the server certificate to its trust store (hopefully after user interaction confirming this).

The client then responds with its key-exchange response, and its own certificate if applicable (which it seldom is). It immediately follows that with its `Finished` message, the first message that is encrypted with the just-negotiated cipher suite. The server then replies with *its* encrypted `Finished` message, and encrypted application-layer communication can then begin.

Most browsers allow the user to click on the padlock icon for the TLS-secured connection to find out what cipher suite was actually agreed upon.

A client starting a brand-new connection leaves the Session ID field empty in the `ClientHello` message. However, if a client wishes to resume a previous session, it includes here the Session ID from that previous session. The server must, of course, also recognize that session.

Once upon a time, some broken TLS servers failed to respond properly if a client proposed a version number greater than what the server supported; the server would close the connection instead of returning a lower version number. As a result, many clients were programmed to try again with the next-lower version in the event of *any* connection-setup failure. This meant that an attacker could force a client to propose an obsolete version (*eg* SSL 3.0) simply by interrupting earlier connection-setup attempts, perhaps with a TCP RST packet. A consequence was the POODLE vulnerability mentioned in the sidebar in *22.10 SSH and TLS*.

An application's choice of TLS (versus unencrypted communication) is often signaled by the use of a special port number; for example, the standard http port is 80 while the standard https port is 443. Alternatively, there may be some initial negotiation; for example, in the SMTP email protocol (**RFC 5321**) it is common for the client to connect to the server on the standard port 25 without encryption, but then to negotiate the use of TLS using the STARTTLS extension to SMTP (**RFC 3207**). This adds multiple extra RTTs, but for non-interactive protocols this is usually of only minor concern.

Occasionally applications do sometimes get confused about whether TLS is in use. Web browsers connecting to port 80 (HTTP) or port 443 (HTTPS) can usually figure things out correctly, even if the "http://" or "https://" part of the URL is missing. However, browsers connecting to nonstandard webservers running on nonstandard ports may receive a very cryptic binary response if TLS was required but the "https://" URL prefix was missing.

22.10.2.4.1 Domain Fronting

Government censorship of websites and other Internet services – such as encrypted messaging – is rampant in many parts of the world. We saw above that the ServerName extension is included in many `ClientHello` messages in order to request the correct certificate. This is sent in the clear, and can be used by censors to block selected traffic.

One censorship workaround, known as domain fronting, is to ask for one hostname, say `alice.org`, in the `ClientHello` ServerName extension, but then request a second, different hostname, `badbob.net`, within the application-layer protocol, after encryption is in place. For example, an HTTP GET request (*12.6.2 netcat again*) almost always contains a `HOST:` header; this is where `badbob.net` would be specified. Of course, this only works if `alice.org` and `badbob.net` are hosted at the same IP address, but large HTTP servers often do handle multiple websites on a single server using a mechanism known as virtual hosting (*7.8.1 nslookup (and dig)*). To block `badbob.net`, a censor must now also block `alice.org`. The hope is that censors will be unwilling to do that.

Domain fronting is even easier when CDNs (*1.12.2 Content-Distribution Networks*) are involved; virtual hosting is not necessary. A CDN accepts connections to the websites of a large number of customers (often a much larger number than a single server can support using virtual hosting) a each *edge server* of their network. The edge servers have certificates for each of the customers, and so appear legitimate to users. After the connection, the edge server might then answer some requests with cached data, but will forward other requests on to the actual web server. To enable domain fronting in this environment, `badbob.net` might only need to become a customer of the same CDN as `alice.org`.

CDNs and large hosting providers sometimes do not think highly of domain fronting, particularly as some censors have shown willingness to block critical domains like `google.com` just to block an unwanted application. Google and Amazon Web Services disabled domain fronting for their CDN customers in 2018.

Before the initial connection to the server, the client will usually make a DNS request to obtain the server's IP address. These too can be blocked, or monitored, although there are workarounds.

22.10.2.4.2 TLS key exchange

Perhaps the most important part of the TLS handshake is to negotiate the encyption keys. The keys themselves are all derived from the TLS **master secret**. The key derivations are done in deterministic ways (*eg* using secure hashes, or a stream-cipher algorithm), and can be done by either side once the master secret is known.

In all cases, the client should know the master secret after receiving the `ServerHello` and related packets. At this point – after one RTT – the client can begin sending encrypted messages to the server. At this point, the client can even begin sending encrypted data, although it is possible that the client is not yet fully authenticated to the server.

Negotiation of the master secret depends on the cipher suite chosen. In the **RSA key-exchange** method, the client chooses a random **pre-master secret**. This is sent to the server in the third leg of the exchange, after the client has received the server's certificate in the `ServerHello` stage. The pre-master secret is encrypted with the public RSA key from the server's certificate, thus conveying it securely to the server. Both sides calculate the master secret from the pre-master secret and both sides' cryptographic nonces, using a secure hash or the pseudorandom keystream of a stream cipher (*22.7.4 Stream Ciphers*). The inclusion of the server nonce means tha the client does not unilaterally specify the key.

If **Diffie-Hellman-Merkle** key-exchange is used (*22.8 Diffie-Hellman-Merkle Exchange*), then the server proposes the prime p and primitive root g during the `ServerHello` phase, as part of its selection of one of the cipher suites listed in the `ClientHello`. The server (Bob) also includes g^b mod p. The client chooses its exponent a, and sends g^a mod p to the server. The pre-master secret is $g^{a \times b}$ mod p, which the client, as in the RSA case, knows at the end of the first RTT. The client's g^a mod p cannot be encrypted using the master key, but everything else sent by the client at that point can be. There is also a version of Diffie-Hellman-Merkle exchange that uses elliptic-curve cryptography.

22.10.2.4.3 TLS version 1.3

Version 1.3 of TLS, **RFC 8446**, makes numerous improvements to the protocol. Among other things, it prunes the list of acceptable cipher suites of all algorithms no longer considered secure, and also of all key-exchange algorithms that do not support forward secrecy, *22.9.2 Forward Secrecy*. The version-negotiation process has also been improved, to prevent version-downgrade attacks, and the 0-RTT mode (below) enables faster connection startup.

Of the two key-exchange methods in the previous section, Diffie-Hellman-Merkle does support forward secrecy while RSA may not, depending on the latter's choice of the pre-master secret. The proposal to drop RSA key exchange has caused concern in some enterprise settings, particularly the banking industry, where widespread use is often made of middleboxes that are able to decrypt and monitor internal TLS connections (that is, connections where both endpoints are under control of the same enterprise). This monitoring is done for a variety of reasons, including regulatory compliance and malware detection.

To enable such monitoring using TLS v1.2, all TLS connections between the enterprise and the outside Internet terminate at the site's border, at a special TLS-proxy appliance. From there, connections to interior systems are re-encrypted, but using a static RSA pre-master secret that is shared with the monitoring equipment. The same applies to any TLS connections that are entirely internal to the enterprise.

To enable this kind of monitoring under TLS v1.3, one approach is to have the monitoring occur at the TLS-proxy appliance, though this means that device will have to be considerably bigger, more complicated and more expensive; additionally, it will not help with entirely internal TLS connections. Another approach is to have the TLS proxy, and at least one endpoint of internal TLS connections, turn over all session keys to the monitors. This raises other security concerns. For two proposed amendments to the original TLS v1.3 proposal, see the Internet Drafts draft-rhrd-tls-tls13-visibility and draft-green-tls-static-dh-in-tls13.

TLS v1.3 requires that all messages after the `ServerHello` be encrypted (except possibly one last client-to-server key-exchange message that should be intrinsically secure). The client knows the master secret at this point, and the server will know it as soon as it hears from the client.

22.10.2.4.4 TLS v1.3 0-RTT mode

Finally, TLS v1.3 adds a so-called **O-RTT** mode, in which the client can send encrypted messages from the very beginning, assuming an appropriate master secret has been negotiated during a *previous* connection.

For HTTPS interactions, this is a common case. The 0-RTT mode, in particular, allows the client to send secure *data* along with its `ClientHello`. The server can respond with 0-RTT-protected data along with its `ServerHello`, if it chooses to. After that, everything should be protected with the new session's master secret; this is known as **1-RTT** protection.

To send 0-RTT data, the client must resume the earlier session by including the previous Session ID in its `ClientHello` message. The server *can* refuse 0-RTT data, and demand 1-RTT protection, but usually will not.

The advantage of 0-RTT data is that, if the server accepts it, an answer can return to the client in a single RTT. This is particularly significant in QUIC (*11.1.1 QUIC* and *12.22.4 QUIC Revisited*), in which the TLS handshake is carried out in parallel with the QUIC connection handshake (a replacement for the TCP three-way handshake). This means that the response comes just one RTT after the very first message from the client. QUIC requires TLS v1.3 (or later).

0-RTT protection is not quite as strong ast 1-RTT protection. For one, forward secrecy does not apply. More seriously, a participant receiving 0-RTT data is vulnerable to replay attacks. An attacker cannot modify a previously intercepted 0-RTT message (without breaking the cipher), but can resend it.

At a minimum, the recipient of a 0-RTT request should accept it only if it is **idempotent**: yielding the same results and side-effects whether executed once or twice (*11.5.2 Sun RPC*). This is generally automatic for simple HTTP GET requests. Idempotency is not itself a security guarantee – after all, the request "delete all files" is idempotent. The point, however, is that the particular request must, if part of a replay attack, have been executed before, and if it was safe once, it *should* be safe twice. A TLS server can also suppport other anti-replay mechanisms, such as a database of past requests.

Another consequence of mixing 0-RTT and 1-RTT data is that the recipient needs to be able to tell which requests received 0-RTT protection and which received full 1-RTT protection.

22.10.3 A TLS Programming Example

In this section we introduce a simple pair of programs, tlsserver.c and tlsclient.c, that communicate via TLS. They somewhat resemble the simplex-talk program in *12.6 TCP simplex-talk*, except that neither reads from the command line. Instead, the client sends a message to the server (which may ignore it), and then the server sends a message back. This structure will allow us later to point the client at a "real" TLS-using (that is, HTTPS-using) web server, instead of `tlsserver`, have the client send an HTTP GET request (*12.6.2 netcat again*), and obtain the web server's response.

Both programs are written in C, in order to use the OpenSSL library, a descendant of the original Netscape SSL implementation. The openssl package also includes some important command-line utilities for certificate creation. While OpenSSL library documentation remains notoriously spare, parts of the library have now been audited, and OpenSSL remains the most widely used implementation of TLS.

The code shown here is intended simply as a demonstration; **it should not be considered a model of how to implement secure connections**.

22.10.3.1 Making a certificate

The first step is to create the appropriate application certificate, and create our own certificate authority to sign it. For each step we will use the `openssl` command, also used below at *22.13 RSA Key Examples* with additional explanation. We start with the certificate authority. The first step is to create the key our certificate authority will use; this is what the `genrsa` option below achieves. We choose a key length of 4096 bits.

```
openssl genrsa -out CAkey.pem 4096
```

The resultant file says it is a "private key", but it is in fact a public/private key pair.

The next step is to create a **self-signed certificate**. The `req` option asks for a signing "request", the `-new` option indicates this is a new request, and the `-x509` option tells openssl to forget making a "request" and instead make a self-signed certificate. X.509 is actually a format standard.

```
openssl req -new -x509 -key CAkey.pem -out CAcert.pem -days 10000
```

The certificate here is set to expire in 10,000 days; real *published* certificates are not supposed to be valid for more than about three years. The signing process (self- or not) triggers a series of questions that will form the "Distinguished Name" of the certificate:

```
Country Name (2 letter code): US
State or Province Name (full name): Illinois
Locality Name (eg, city):Shabbona
Organization Name (eg, company): An Introduction to Computer Networks
Organizational Unit Name (eg, section): Security
Common Name (e.g. server FQDN or YOUR name): Peter L Dordal
Email Address []:
```

If we were requesting a certificate for a web server, we would use the hostname as Common Name, but we are not. The output here, CAcert.pem, represents the concatenation of the above information with the public key from CAkey.pem, and then signed by the private key from CAkey.pem. The private key itself, however, is *not* present in CAcert.pem; this is the file that the TLS client will receive as its certificate authority. We can read the information from the file as follows:

```
openssl x509 -in CAcert.pem -text
```

This produces the following, somewhat edited for space. We can see that the certificate is self-signed because the `Issuer:` is the same as the `Subject:`

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 14576542390929842281 (0xca4a481320cbe069)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=Illinois, L=Shabbona, O=An Introduction to Computer␣
↪Networks, OU=Security, CN=Peter L Dordal
        Validity
            Not Before: Jan 10 20:17:55 2017 GMT
            Not After : May 28 20:17:55 2044 GMT
        Subject: C=US, ST=Illinois, L=Shabbona, O=An Introduction to Computer␣
↪Networks, OU=Security, CN=Peter L Dordal
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:b0:70:06:7e:38:1d:29:35:a7:ca:40:bf:fd:6e:
                    e5:26:7b:ee:0d:e7:d7:c2:61:8e:42:5f:b9:85:8c:
                    ...
                    d4:12:cf
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                D1:74:15:F5:31:CB:DD:FA:D6:AE:81:7A:40:AA:64:7A:55:96:2E:08
            X509v3 Authority Key Identifier:
                ␣
↪keyid:D1:74:15:F5:31:CB:DD:FA:D6:AE:81:7A:40:AA:64:7A:55:96:2E:08

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
         2a:d2:35:43:c2:5d:1c:5d:e2:88:ed:4e:aa:d2:b5:d6:e9:26:
         60:f0:37:ea:29:56:14:62:58:01:78:b0:6f:ee:ab:40:17:36:
         ...
         eb:3d:da:79:5c:90:4d:c9
-----BEGIN CERTIFICATE-----
MIIF8TCCA9mgAwIBAgIJAMpKSBMgy+BpMA0GCSqGSIb3DQEBCwUAMIGOMQswCQYD
VQQGEwJVUzERMA8GA1UECAwISWxsaW5vaXMxETAPBgNVBAcMCFNoYWJib25hMS0w
...
ywhRBNEO1XXB7bFrkkv93q4G3Re2zyw2/5BDEn7rPdp5XJBNyQ==
-----END CERTIFICATE-----
```

We now create the application key and then the application **signature request**. This request we will then sign with the above certificate CAfile.pem to generate the application certificate.

```
openssl genrsa -out appkey.pem 2048
openssl req -new -key appkey.pem -out appreq.pem -days 10000
```

The certificate request here again requires entering a Distinguished Name. This time we enter as follows; the Organization Name must match the CAcert.pem above:

```
Country Name (2 letter code): US
State or Province Name (full name): Illinois
Locality Name (eg, city): no fixed abode
Organization Name (eg, company): An Introduction to Computer Networks
Organizational Unit Name (eg, section): TLS server
Common Name (e.g. server FQDN or YOUR name): Odradek
Email Address []:
```

We are also asked for a "challenge password", but we leave this blank.

Now we come to the final step: the actual signing. Unlike any of the previous openssl commands, this requires root privileges. It also makes use of the global openssl configuration file (/etc/ssl/openssl.cnf), which, among other things, references a file "serial" to assign the certificate a serial number. (We did not have to do any of this to create the *self*-signed certificate.)

```
openssl ca -in appreq.pem -out appcert.pem -days 10000 -cert CAcert.pem -
↪keyfile CAkey.pem
```

We now have our application certificate in appcert.pem, which we deliver to the application section, below. If we read it with `openssl x509 -in appcert.pem -text`, we find that the Issuer is that of CAfile.pem, but the Subject is as above, with Common Name = Odradek.

22.10.3.2 TLSserver

The complete server is at tlsserver.c, along with the certificate and key files appcert.pem and appkey.pem (both stored with an additional `.text` suffix to prevent an accidental click from loading them directly into a browser). To compiler the server under Linux, with the OpenSSL library installed in the usual place, we use

```
gcc -o tlsserver tlsserver.c -lssl -lcrypto
```

The server should be run in the same directory with its certificate and key files.

In the following we go through the important lines of the full program stripped of any error checking. Most OpenSSL library methods return 1 on success and 0 on failure. Different kinds of failure may require different error-message libraries.

```
SSL_library_init();
```

This does what it sounds like: initializes the SSL subsystem. Loading error strings may also be useful. The next steps commit us to the versions of SSL we agree to accept.

```
method = SSLv23_server_method();
ctx = SSL_CTX_new(method);
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1␣
↪| SSL_OP_NO_TLSv1_1);
```

Somewhat paradoxically, `SSLv23_server_method()` accepts *all* SSL and TLS versions. In the third line above, we then disable everything earlier than TLS version 1.1. In openssl version 1.1.0 (the numbering is unrelated to the TLS versioning), `SSLv23_server_method()` can be replaced with the more appropriately named `TLS_server_method()`.

The variable `ctx` represents a TLS *context*, which is a set of TLS state information. Our server will use the same context for all incoming connections.

Next we load the server certificate and key files into our newly created TLS context. Recall that the server side gets our application certificate `appcert.pem`; the client side will get our certificate-authority certificate `CAcert.pem`.

```
int cfile_result = SSL_CTX_use_certificate_file(ctx, "./appcert.pem", SSL_
↪FILETYPE_PEM);
int kfile_result = SSL_CTX_use_PrivateKey_file (ctx, "./appkey.pem",  SSL_
↪FILETYPE_PEM);
```

The server needs the key file to sign messages. Next we create an ordinary TCP socket listening on port 4433; `createSocket()` is defined at the end of the tlsserver.c file.

```
sock = createSocket(4433);
```

Finally we get to the main loop. We accept a TCP connection, create a new connection-specific SSL object from our context, and tie the new SSL object to the socket.

```
while(1) {
    int childsock = accept(sock, (struct sockaddr*)&addr, &len);
    SSL* ssl = SSL_new(ctx);
    SSL_set_fd(ssl, childsock);
    SSL_accept(ssl);
    SSL_write(ssl, reply, strlen(reply));
}
```

`SSL_accept()` is where the handshaking described in *22.10.2.4 TLS Connection Setup* takes place. At this point, the server writes its `reply` message over the now-encrypted channel, and is done.

We argued in *1.15 IETF and OSI* that TLS can in some ways be regarded as a true Presentation layer. Note, however, that the Application layer here (that is, tlsserver.c) is responsible for accepting `childsock`, and passing it to TLS through `SSL_set_fd()`; the Application layer, in other words, does interact directly with TCP.

22.10.3.3 TLSclient

The complete client is at tlsclient.c; its certificate is CAcert.pem (again with a `.text` suffix). Again we go through the sequence of SSL library calls with error-checking removed. As with the server, we start with initialization; this time, we also load error strings:

```
SSL_library_init();   // "SSL_library_init() always returns '1'
ERR_load_crypto_strings();
SSL_load_error_strings();
```

Next we again choose what TLS versions we will allow, this time starting with `SSLv23_`**client**`_method()`:

```
method = SSLv23_client_method();
ctx = SSL_CTX_new(method);
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3|SSL_OP_NO_
→TLSv1|SSL_OP_NO_TLSv1_1);
```

If we instead use `method = TLSv1_1_client_method()`, the connection should fail, this call allows *only* TLS version 1.1, and the server requires TLS version 1.2 or better.

The next step is to load the **trust store**, that is, the certificates from the certificate authorities we have elected to trust. If we do nothing, the trust store will be empty. We first load a standard certificate directory (directories are supplied to `SSL_CTX_load_verify_locations()` as the third parameter and individual files as the second). Certificates in a directory must be named (possibly using symbolic links) by their hash values; see the c_rehash utility. If all we wanted was to be able to trust our own server's `appcert.pem`, we could just load our own certificate-authority certificate CAcert.pem, but we will need the standard certificate directory if we want to point tlsclient at a real HTTPS server rather than at tlsserver.

```
SSL_CTX_load_verify_locations(ctx, NULL, "/etc/ssl/certs")
```

Next we load our own certificate CAcert.pem, which is then *added* to the trust store, in addition to the standard certificates. We can add multiple individual certificates by making multiple calls, or by concatenating all the certificates into a single file. The certificates must be separated by their BEGIN CERTIFICATE and END CERTIFICATE lines.

```
SSL_CTX_load_verify_locations(ctx, "CAcert.pem", NULL);
```

Now we're ready to connect to the server. The last line below initiates the TLS handshake, starting with ClientHello.

```
sock = openConnection(hostname, port);
ssl = SSL_new(ctx);
SSL_set_fd(ssl, sock);
SSL_connect(ssl);
```

Next the client retrieves (and, in our case prints) the certificate supplied by the server. If the server is tlsserver, this will normally be appcert.pem, with CN=Odradek.

```
cert = SSL_get_peer_certificate(ssl);
certname = X509_get_subject_name(cert);
// print certname
```

The printed certname does indeed show CN=Odradek, from appcert.pem. If we were writing a web browser, this is the point where we would verify that the site hostname matches the CN field of the certificate.

After the client receives the application certificate, it must **verify** its signature, with the call below. This is where the client uses its trust store.

```
ret = SSL_get_verify_result(ssl);
```

If one of the certificate authorities in the trust store vouches for the signature on the application certificate, the return value above is X509_V_OK, and all is well. If we comment out the loading of CAcert.pem, however, we get "unable to get local issuer certificate". If, with tlsclient still not loading CAcert.pem, we have the *server* send CAcert.pem (and CAkey.pem), instead of its proper certificate, we get an error "self-signed certificate". A full list of certificate-verification errors is listed with the verify command.

If validation fails, the connection is still encrypted, but is vulnerable to a man-in-the-middle attack. Regardless of what happened during validation, our particular tlsclient goes on to write an HTTP GET request to the server, and then read the server's response (the companion tlsserver program does not in fact read the GET request). Generally speaking, however, **continuing the TLS session after a certificate validation failure is a very bad idea**.

```
SSL_write(ssl, request, req_len);      // ignored by tlsserver program
do {
    bytesread = SSL_read(ssl, buf, sizeof(buf));
    fwrite(buf, bytesread, 1, stdout);
} while(bytesread > 0);
```

The written request here is ignored by `tlsserver`; it is an HTTP GET request of the form `GET / HTTP/1.1\r\nHost:` *hostname*`\r\n\r\n`. If we point `tlsclient` at a real webserver, say

```
tlsclient google.com 443
```

then we should again get an `X509_V_OK` verification result because we loaded the default certificate-authority library.

We can also point the built-in openssl client at `tlsserver`; by default it connects to `localhost` at port 4433:

```
openssl s_client
```

Of course, verification fails. This is because `s_client` doesn't know about our certificate authority. We can add it, however, on the command line:

```
openssl s_client -CAfile CAcert.pem
```

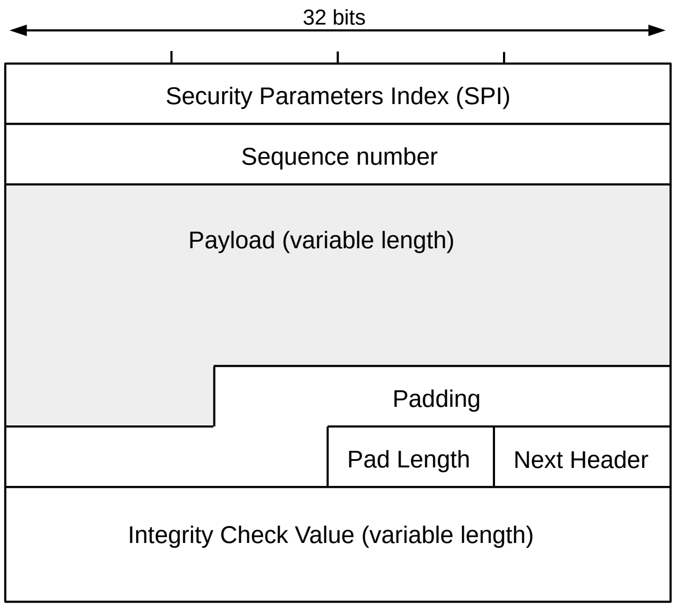
Now the verification is successful.

22.11 IPsec

The SSH software package was built from the ground up to implement the SSH protocol. All modern web browsers incorporate TLS libraries to enable secure web connections. What can you do if you want to add encryption (or authentication) to a network application that doesn't have it built in? Or, alternatively, how can you as a system administrator ensure that everyone's traffic is protected, regardless of what software they are using?

IPsec, for "IP security", is one answer. It is a general-purpose security protocol which typically behaves as if it were a network sublayer *below* the IP layer (or, in transport mode, below the Transport layer). In this it is akin to Wi-Fi (*3.7.5 Wi-Fi Security*), which implements encryption within the LAN layer; in both Wi-Fi and IPsec the encryption is transparent to the communicating applications. In terms of actual implementation it is most often incorporated within the IP layer, but can be implemented as an external network appliance.

IPsec can be used to protect anything from individual TCP (or UDP) connections to all traffic between a pair of routers. It is often used to implement VPN-like access from "outside" hosts to private subnets behind NAT routers. It is easily adapted to support any encryption or authentication mechanism. IPsec supports two packet formats: the **authentication header**, AH, for authentication only, and the **encapsulating security payload**, ESP, below, for either authentication or encryption or both. The ESP format is much more common and is the only one we will consider here. The AH format dates from the days when most export of encryption software from the United States was banned (see the sidebar 'Crypto Law' at *22.7.2 Block Ciphers*), and, in any event, the ESP format can be used for authentication only. The ESP packet format is as follows:


32 bits
Security Parameters Index (SPI)
Sequence number
Payload (variable length)
Padding
Pad Length
Next Header
Integrity Check Value (variable length)

ESP packet layout

The SPI identifies the security association, below. The sequence number is there to prevent replay attacks. Senders must increment it on every transmission, but receivers care only if the received numbers are not strictly increasing; gaps due to lost packets do not matter. The cryptographic algorithm applied to the payload and the integrity-check algorithm are negotiated at connection set-up. The Padding field is used first to bring the Payload length up to a multiple of the applicable encryption blocksize, and then to round up the total to a multiple of four bytes. The Next Header field describes the data that is *inside* the Payload, *eg* TCP or UDP for Transport mode or IP for Tunnel mode. It corresponds to the Protocol field of *7.1 The IPv4 Header* or the Next Header field of *8.1 The IPv6 Header*.

IPsec has two primary modes: **transport** and **tunnel**. In transport mode, the IPsec endpoints are also typically the traffic endpoints, and only the transport-layer header (*eg* TCP header) and data are encrypted or protected. In the more-common tunnel mode one of the IPsec endpoints is often a router (or “security gateway”); encryption or protection includes the original IP headers, so that an eavesdropper cannot necessarily identify the actual traffic endpoints.

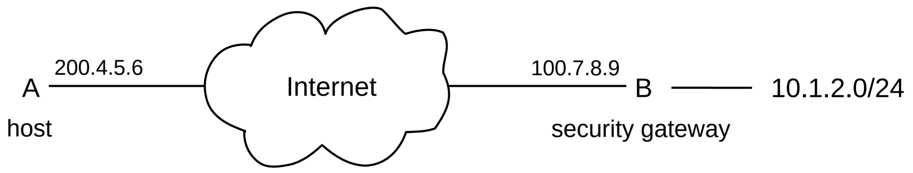
IPsec is documented in a wide range of RFCs. A good overview of the architectural principles is found in **RFC 4301**. The ESP packet format is described in **RFC 4303**.

A word of warning: while IPsec does support modern encryption, it also continues to support outdated algorithms as well; users must take care to ensure that the encryption negotiated is sufficient. IPsec has also attracted, in recent years, rather less attention from the security community than SSH or TLS, and “many eyes make all bugs shallow”. Or at least some bugs.

22.11.1 Security Associations

In order for a given connection or node-to-node path to receive IPsec protection, it is first necessary to set up a pair of **security associations**. A security association consists of all necessary encryption/authentication attributes – algorithms, keys, rekeying rules, *etc* – together with a set of **selectors** to identify the covered traffic. A given security association covers traffic in one direction only; bidirectional traffic requires a separate security association for each direction. For outbound traffic – that is, traffic going from unprotected (internal) to IPsec-protected status – the selector consists of the destination IP address (or set of addresses) and possibly also the source IP address (or set of source addresses) and port or protocol values. Inbound ESP packets carry a 32-bit Security Parameters Index, or SPI, that for unicast traffic identifies the security association. However, that security association must still be checked against the packet for an actual match.

The destination and source IP addresses need not be the same as the IP addresses of the IPsec endpoints. As an example, consider the following tunnel-mode arrangement, in which traveling host A wants to connect to private subnet 10.1.2.0/24 through security gateway B. IPv4 addresses are shown, but the same arrangement can be created with IPv6.



The A-to-B IPsec security association's selector will include the entire subnet 10.1.2.0/24 in its set of destination addresses. A packet from A to 10.1.2.3 arriving at A's IPsec interface will match this selector, and will be encapsulated and sent (via normal Internet routing) to B at 100.7.8.9. B will de-encapsulate the packet, and then forward it on to 10.1.2.3 using its normal IP forwarding table. B might actually *be* the NAT router at its site, with external address 10.7.8.9 and internal subnet 10.1.2.0/24, or B might simply be a publicly visible host at its site that happens to have a route to the private 10.1.2.0/24 subnet.

The action of forwarding the encapsulated packet from A to B closely resembles IP forwarding, but isn't quite. It is unlikely A will have a true forwarding-table entry for 10.1.2.0/24 at all; it will very likely have only a single default route to its local ISP connection. Delivery of the packet cannot be understood simply by examining A's IP forwarding table. A might even have a forwarding-table entry for 10.1.2.0/24 to somewhere else, but the IPsec "pseudo-route" to B's 10.1.2.0/24 is still the one taken. This can easily lead to confusion; for complex arrangements with multiple overlapping security associations, this can lead to nontrivial difficulties in figuring out just how a packet is forwarded.

A second routing issue exists at B's end. Host 10.1.2.3 will see the packet from A arrive with address 200.4.5.6. Its reply back to A will be delivered to A using the tunnel only if the B-site routing infrastructure routes the packet back to B. If B is the NAT router, this will happen as a matter of course, but otherwise some deliberate action may need to be taken to avoid having 10.1.2.3-to-A traffic take an unsecured route. Additionally, the B-to-A security association needs to list 10.1.2.0/24 in its list of *source* addresses. The IPsec "pseudo-route" now resembles the policy-based routing of *9.6 Routing on Other Attributes*, with routing based on both destination and source addresses. A packet for A arriving at B with source address 10.**2.4**.3 should *not* take the IPsec tunnel. (To add confusion, Linux IPsec pseudo-routes do not actually show up in the Linux policy-based routing tables.)

Security associations are created through a software **management interface**, *eg* via the Linux ipsec command and the associated configuration file ipsec.conf. It is possible for an application to request creation of the necessary security associations, but it is more common for these to be set up *before* the IPsec-protected application starts up.

A request for the creation of a security association typically triggers the invocation of the **Internet Key Exchange**, IKE, protocol; the current version 2 is often abbreviated IKEv2. IKEv2 is described in **RFC**

7296. IKEv2 typically uses public keys to negotiate a session key (*22.7.1 Session Keys*); IKEv2 may then renegotiate the session key at intervals. In the simplest (and not very secure) case, both sides have been manually configured with a session key, and IKEv2 has little to do beyond verifying that the two sides have the same key.

NAT traversal of IPsec packets is particularly tricky. For AH packets it is impossible, because the cryptographic authentication code in the packet covers the original IP addresses, as well as the packet transport data. That is not an issue for ESP packets, but even there the incoming packet must match the receivers's security-association selector, which it will not if that was negotiated using the sender's original IP address. An additional problem is that many NAT routers fail to forward (or fail to forward properly) packets outside of protocols ICMP, UDP and TCP.

As a result, IPsec has its very own NAT-traversal mechanism, outlined in **RFC 3715**, **RFC 3947** and **RFC 3948**. IPsec packets are encapsulated in UDP packets, with their original headers. After de-encapsulation at the IPsec receiving end, it is these original headers that are used in the security-association check. Additionally, a keepalive mechanism is defined in which the IPsec nodes send regular small packets to make sure the NAT mapping for the connection does not time out.

22.12 DNSSEC

The DNS Security Extensions, DNSSEC, make it possible for authoritative nameservers to provide authenticated responses to DNS queries, by using **digital signatures**, below. The primary goal of DNSSEC is to prevent cache poisoning (*7.8.3 DNS Cache Poisoning*) by allowing resolvers to verify any DNS records received.

DNSSEC is documented in **RFC 4033**, **RFC 4034**, **RFC 4035** and updates. **RFC 6891** outlines a general framework for extensions to DNS, known as EDNS; these extensions include new record types. EDNS in particular defines the DNSSEC OK, or DO, flag; this is used to signal that the receiving nameserver should return a DNSSEC-aware response.

The basic idea behind DNSSEC is for each zone, including the root zone, to digitally sign all of its RRsets – sets of DNS resource records matching a given name (*eg* `cs.luc.edu`) and type (*eg* A records, AAAA records, or MX records) – with a public-key signature. Each zone *except* the root zone then has its parent zone sign its public key. The root-zone public key must be pre-loaded into the resolver or end-system, much as the root-zone IP addresses must be pre-loaded. This sort of linked sequence of digital-signature verifications is often called a **chain of trust**. The root keys are known as the **trust anchors**; these can be compared to **certificate authorities**, *22.10.2.1 Certificate Authorities*.

The root zone implemented support for DNSSEC in 2010, after .gov and with the .org zone not far behind; other top-level domains including .com, .net and .edu became DNSSEC-aware by the following year.

In order to support all these signed records and keys, DNSSEC introduced several additional DNS record types:

- `RRSIG`: a signature for an RRset.
- `DNSKEY`: the public half of the keypair used to sign zone records.
- `DS`: a "delegation signer" record in a parent zone containing a signature for a child zone's zone-signing key.

- `NSEC` and `NSEC3`: the next DNS name in sequence, used to affirm securely that a given hostname does *not* exist.

In slightly more detail, each DNSSEC authoritative nameserver creates, for each zone, a Zone Signing Key, or ZSK. For each possible RRset the nameserver creates an `RRSIG` signature record, signed by the ZSK. The public half of the zone's ZSK is available in a record of type `DNSKEY`, and with DNS name equal to the zone name. Finally, a hash of the public ZSK is signed by the *parent*'s ZSK and made available in the parent zone as a record of type `DS`, again with name equal to the child zone's name.

If a DNSSEC-aware query arrives for ⟨*name*,*type*⟩, the nameserver will return both that RRset and also the RRSIG record for that name and type. For example, if we use the dig tool to query for the `A` record for `isc.org`, and include the `+dnssec` flag for DNSSEC-aware results, `dig isc.org A +dnssec`, we receive (in part, and with formatting applied)

```
isc.org.         52       IN       A        149.20.64.69
isc.org.         52       IN       RRSIG    A 5 2 60 20190328000617␣
↪20190226000617 19923 isc.org.
                                            ␣
↪KrBysOlbe4L6sJJOJNbJhfAuNt11q+6A2cQTnr3CXeFwxYJTXdqAkSwg
                                            ␣
↪QzGHpIrVfOw2dn6GdqXQ6umqU1cnFNtXumdvUp45+XSCoZC6YciR4xNs
                                         f8YMR5F66LIcMZewP11ofWOV6/
↪m9rSfR38FRnDkPf3Jg+O2+qvSKQ+Mq
                                         lV8=
```

Note that RRSIG records must always be piggybacked onto the original query, as there is no independent way to request "RRSIG records matching type *type*". DNS queries may contain only one type. The RRSIG records are part of the DNS ANSWER section, not the ADDITIONAL section; these records are an essential part of the response.

We can obtain the `isc.org` zone's public ZSK by requesting the record with name `isc.org` and type DNSKEY. We can verify this key by asking for the record with name `isc.org` and type DS from the `.org` nameserver.

To fetch the A record for `www.example.com`, a DNSSEC resolver (often called a *validating* resolver) would (if nothing has been cached) start by asking the root zone for the appropriate NS record, just as with plain DNS (*7.8.1 nslookup (and dig)*). The root zone replies with the NS record (and A record) pointing to the `.com` authoritative nameserver, and, because the DNSSEC OK bit in the request has been set, also includes the corresponding RRSIG record. The resolver can validate the RRSIG signature because it knows by prearrangement the root-zone KSK.

The resolver also asks the root nameserver for the DS record for `.com`. This is a signature for the KSK that the `.com` nameserver uses, signed by the root key. It will be used in the next step.

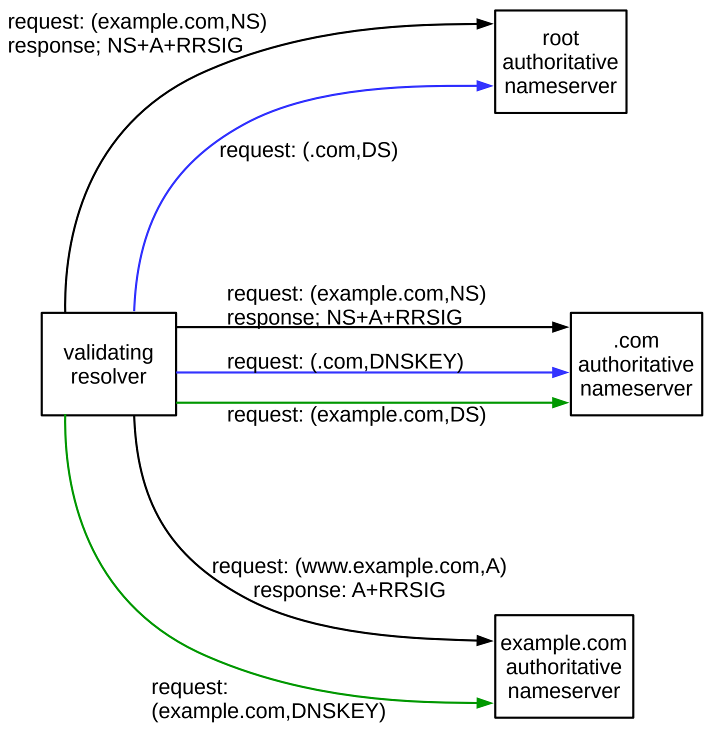
The resolver now switches to sending its requests to the authoritative nameserver for the `.com` zone. It asks for the NS record for `example.com`, and receives the appropriate NS and A records. Because DNSSEC is involved, it also receives the corresponding RRSIG records. The latter are signed with the KSK for `.com`. The resolver obtains this KSK by requesting the DNSKEY record from the `.com` nameserver. This KSK can then be validated by using the signed DS record previously obtained from the root zone, and the validated KSK can then in turn be used to validate the RRSIG records.

Note that the DNSKEY and DS records for the `.com` DNS name form a pair, with the latter signing the former (or a hash of the former). The DNSKEY record, though, is obtained from the `.com` nameserver, while the corresponding DS record is obtained from the parent (root, in this case) nameserver.

The resolver also asks the `.com` nameserver for the DS record for `example.com`; this will be used in the next step.

In the final stage, the resolver sends its requests to the `example.com` namserver. This time it asks for the A record for `www.example.com`. This address is returned, along with the corresponding RRSIG. The resolver gets the `example.com` ZSK by requesting the DNSKEY record. The resolver obtained a signature for this ZSK in the previous DS record (obtained from the `.com` nameserver), and so can validate this `example.com` ZSK, and so in turn can validate the RRSIG for the address record for `www.example.com`.

The DNS queries involved are summarized in the following diagram. Each request shows the ⟨*name*,*type*⟩ pair. Responses are indicated only for queries that traditional, non-DNSSEC DNS would make, to emphasize RRSIG records are included. These queries are shown with black paths. The DS/DNSKEY requests for the `.com` zone are shown with blue paths; the DS/DNSKEY requests for the `example.com` zone are shown in green.


request: (example.com,NS)
response; NS+A+RRSIG
root authoritative nameserver
request: (.com,DS)
request: (example.com,NS)
response; NS+A+RRSIG
validating resolver
request: (.com,DNSKEY)
request: (example.com,DS)
.com authoritative nameserver
request: (www.example.com,A)
response: A+RRSIG
example.com authoritative nameserver
request:
(example.com,DNSKEY)

The chain of key validations can be summarized as follows:

- The known root ZSK validates the `.com` ZSK, through the DNSKEY/DS pair for the DNS name `.com`
- The `.com` ZSK validates the `example.com` ZSK, and is itself validated via the DNSKEY/DS pair for the DNS name `.com`
- The `example.com` ZSK validates the A record for `www.example.com`, and is itself validated via

the DNSKEY/DS pair for the DNS name `example.com`

It is possible that the resolver knows, likely through caching, the IP address of the `example.com` nameserver. In this case, if the RRSIG and DS credentials were not also cached, the resolver might go through the above process from the bottom up (`example.com` to `.com` to root) to validate the information.

If `example.com` had not yet implemented DNSSEC support, but a subdomain `cs.example.com` did, then the chain of trust between the subdomain and the root zone would be broken. In this case, a resolver could only use DNSSEC to validate records for `cs.example.com` if it already had the ZSK for this domain, known, as with the root zones, as the trust anchor. Trust anchors for such isolated zones, or "islands of security", are often made available through manual configuration.

Any and all of the records above might be cached by the validating resolver, following a previous query. Caching works for DNSSEC just as it works for ordinary DNS. The key-related records, like all DNS records, each have a time-to-live (TTL) value; the resolver must abide by these.

The diagram above shows two separate requests to the root nameserver, three to the `.com` nameserver, and two to the `example.com` nameserver. That is inefficient. Fortunately, it is not mandatory: clients can request everything at once. It is also likely that many of these records will be cached (particularly requests to the root and `.com` nameservers).

For improved security, some authoritative nameservers may be configured with two keys: a shorter zone-signing key, used to sign the RRSIGs, and then a longer **key-signing key**, or KSK; the latter is the one signed in the parent-zone DS record. The DNSKEY records return *both* ZSK and KSK; this is sufficient to maintain the chain of trust.

As an example of a ZSK/KSK pair, let us send the following DNSKEY request to the `.com` domain (having previously looked up the NS record to get the `.com` nameserver IP address, 192.12.94.30)

```
dig @192.12.94.30 com DNSKEY +dnssec
```

We get back two keys. The shorter one, as is explained below, is the ZSK and the longer is the KSK (this output has been formatted so the key data lines up neatly).

```
com.    86400   IN      DNSKEY  256 3 8 AQO+kWUV3rtj/
↪Vi6FLBfxMRcFoz69Go6xVwa99AWzENDi98y9CIJfx6w
                                        n9aR0SWsCk/
↪oY+hreX6egC7nyyxQ5bxq52aovlZI34Cn+hpy/YGGO2HS
                                        b44AWONsjuZTAfGYLBdaJi2Wg+Z0IVqPw/
↪Lp0Ysu9I8orc2KyNIPQGA/
                                        rTgXOw==
com.    86400   IN      DNSKEY  257 3 8 AQPDzldNmMvZFX4NcNJ0uEnKDg7tmv/
↪F3MyQR0lpBmVcNcsIszxNFxsB
                                       ␣
↪fKNW9JYCYqpik8366LE7VbIcNRzfp2h9OO8HRl+H+E08zauK8k7evWEm
                                        u/6od+2boggPoiEfGNyvNPaSI7FOIroDsnw/
↪taggzHRX1Z7SOiOiPWPN
                                       ␣
↪IwSUyWOZ79VmcQ1GLkC6NlYvG3HwYmynQv6oFwGv/KELSw7ZSdrbTQ0H
                                       ␣
↪XvZbqMUI7BaMskmvgm1G7oKZ1YiF7O9ioVNc0+7ASbqmZN7Z98EGU/Qh
                                        2K/
↪BgUe8Hs0XVcdPKrtyYnoQHd2ynKPcMMlTEih2/2HDHjRPJ2aywIpK
```

```
Nnv4oPo/
```

The numbers 256 and 257 immediately following the DNSKEY type label represent a 16-bit flag field. Both have the bit set in the 256 position, indicating the keys are there for zone-signing generally. The longer key also has the bit set in the 1 position; this is the "Secure Entry Point" flag and, rather loosely, indicates that this key is the KSK. More specifically, the SEP flag is used to mark a key for which a DS record will be created in the parent zone. See **RFC 3757** for further details.

After the 256/257 is a 3, and then an 8. The 8 means that the keys and signatures use RSA and SHA-256, as specified in **RFC 5702**.

The keys are encoded in base64 (**RFC 4648**). The first key has 176 encoded bytes (3 lines of 56, plus 8). Decoding it with the Python `b64decode` function in the `base64` library, we get a byte string of length 130. The first byte is 0x01, with seven leading zero bits; the number of bits from the first nonzero bit to the end is 1033. The second has 344 bytes, or 2057 bits after stripping the leading 0-bits. In common parlance, these are 1024-bit and 2048-bit keys respectively. A 1024-bit RSA keylength is not terribly secure, but is meant for relatively short-term use (~30 days). The 2048-bit KSK provides excellent security.

Each RRSIG record contains a `Key Tag` field, consisting of a 16-bit hash of information about the key. This makes it much easier, when there are multiple DNSKEY records, to help figure out which one was used to create the RRSIG signature. The fallback, if necessary, is to re-calculate the signature with each available key, and see which one matches the RRSIG. Multiple DNSKEY records occur when separate ZSK and KSK keys have been created, as above. They also occur anytime a key is in the process of being updated, as in the next paragraph.

To change the KSK for the `example.com` nameserver, the first step is to create the new key, and then to create new RRSIGs for each RRset. A new DNSKEY record is also created. The new KSK public key must then be communicated to the parent zone, in order for the latter to create the corresponding DS record. For a while, the RRSIG and DNSKEY (and DS, at the parent nameserver, if the KSK is being changed) record sets returned will contain records for *both* KSKs; the receiving resolver can use the `Key Tag` field (above) to figure out which key goes with which RRSIG. Eventually the old KSK will expire, *eg* when its TTL is reached, and the older records can be removed.

Signing failure responses for non-existent DNS names is also important, but is a little trickier. First, the RRset in question is empty. An empty set can be signed, but the signed response can now be replayed, perhaps to convince someone later that an existing DNS name is not valid. It would be possible to prevent this by including the non-existent DNS name in the signed response, but that would require that the signing private key be available whenever necessary. One of the goals of the RRset/RRSIG strategy above, however, is to make possible the pre-signing of all record sets, so the actual private key can then be secured offline.

To get around this, the original strategy for authenticating non-existence was to return a record containing the previous legitimate DNS name, and then an **NSEC** record listing the subsequent legitimate DNS name, according to alphabetical order. The NSEC record corresponding to a valid DNS host name is the next valid DNS name in sequence. For example, if a query asked for information about nonexistent host `foo.example.com`, the results returned might be the DNS name `erl.example.com` and then the NSEC record for `erl` indicating that the next record following was `gateway.example.com`. The nameserver can prepare signatures for such records ahead of time for each consecutive pair of DNS names. The NSEC record corresponding to the last valid hostname wraps around to the first, usually the zone name itself.

The drawback to this strategy is that it makes it easy for someone to enumerate all the valid DNS names in a zone, by sequential querying. To prevent this, the NSEC approach was updated to **NSEC3**, which does much the same thing, except that cryptographic hashes of each hostname are returned instead, and the ordering used is that of the hashed values. An attacker can now enumerate the *hashed* values of each DNS name, but this doesn't help discover the actual hostnames without considerable effort. The necessary signatures can, as with the NSEC approach, all be prepared in advance.

22.12.1 Using DNSSEC

If you are managing an authoritative nameserver, *eg* for your own website, enabling DNSSEC takes some deliberate effort. To enable DNSSEC requires configuring the nameserver software to be DNSSEC-aware, creating the keys, and forwarding the appropriate DS record to the parent zone.

For sites that have done this, however, there is no guarantee that the DNSSEC validation benefits will actually be available to a given user workstation; that depends on the resolver the workstation uses. If it supports DNSSEC, then DNS results from DNSSEC-aware authoritative nameservers will be validated according to the DNSSEC process.

Most user workstations are configured to use the site resolver provided by the local ISP. Some of these support DNSSEC; many do not. It is possible to switch to a DNSSEC-validating resolver manually, *eg* a public DNS server, but most users do not do this. Typical workstation "stub resolvers" do not perform DNSSEC validation by default; the popular Linux dnsmasq resolver can be configured to use DNSSEC by adding the `--dnssec` command-line flag in the appropriate startup file.

Does dnssec-failed.org exist?

If your site DNS resolver performs DNSSEC validation, clicking on the link here to www.dnssec-failed.org will simply fail. To verify the site actually exists without changing resolvers, first get the NS record for `dnssec-failed.org` from the `.org` nameserver; as of 2019 this was dns101.comcast.net at 69.252.250.103. Then `dig @69.252.250.103 www.dnssec-failed.org` should give you the desired A records (two in 2019: 68.87.109.242 and 69.252.193.191).

The simplest way to tell if a resolver supports DNSSEC, *and* validates the DNSSEC responses received is to look up one of several DNS names that have intentionally been misconfigured. One of these is www.dnssec-failed.org, managed (2019) by Comcast. A validating resolver will return the NXDOMAIN (Non-eXistent DOMAIN) error message (or, in other words, zero records in the ANSWER section); clicking on the link should yield a browser error message like "Hmm. We're having trouble finding that site." A non-DNSSEC-validating resolver will return an A record, and the link should work normally. Some partially DNSSEC-aware (but non-validating) resolvers still manage to return an address (see below).

Another way to gauge the degree of resolver support for DNSSEC is to use the dig command(*7.8.1 nslookup (and dig)*). In the example below, the *dns_server* should be the IP address of the resolver; if omitted (along with the `@` sign) then the default resolver is used. The `+dnssec` argument sets the `DNSSEC OK` flag in the query.

```
dig @dns_server isc.org A +dnssec
```

This command returns something like the following if the resolver does not support DNSSEC at all (this is from the OpenDNS resolver at 208.67.222.222, as of 2019).

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 15612
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;isc.org.                        IN      A
;; ANSWER SECTION:
isc.org.                60       IN      A       149.20.64.69
```

There is no RRSIG record, the `flags` on the second line do *not* include the `ad` flag (for "authenticated data"), and the EDNS line does not include the `do` flag (for "DNSSEC OK").

For resolvers understanding DNSSEC, the following might appear (this is from the Google resolver at 8.8.8.8):

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60302
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 512
;; QUESTION SECTION:
;isc.org.                        IN      A
;; ANSWER SECTION:
isc.org.                59       IN      A       149.20.64.69
isc.org.                59       IN      RRSIG   A 5 2 60 20190403233441␣
 ↪20190304233441 28347 isc.org. KlUPLww/FO3rOV3rDiuEH2ttUDSA/
 ↪PtpCaHTGTSyHMaIGlOU3YoHn8hP FJtCkhoopF/␣
 ↪Z6dFXPxAq1LpwtEP9KqU+PxQYGSsZESg1KkEWIaoiE2MB␣
 ↪hHsGqDnJG2GJhyBfCESIt21QY9Q28+nraTG0OUHiKwE8H2c0/PM2VTXP clQ=
```

Note the RRSIG record, the `ad` flag in the second line, and the `do` flag in the fourth.

Without the `+dnssec` argument, we should get an ordinary DNS response, without RRSIG, `ad` or `do`.

Occasional intermediate results may also sometimes appear, *eg* having the RRSIG record and the `do` flag but missing the `ad` flag.

If we try the same lookup with the domain name `www.dnssec-failed.org`, some resolvers return no answer (that is, NXDOMAIN) while others return A and RRSIG records. However, we should not see any resolver return with the `ad` flag, as due to an artificially induced key error the data cannot in this case be authenticated.

22.12.2 DNS-based Authentication of Named Entities

The idea behind DNS-based Authentication of Named Entities, or **DANE**, is to have DNSSEC sign not just the usual DNS records, but also TLS certificates. By doing so, DANE replaces the need for TLS certificate authorities (*22.10.2.1 Certificate Authorities*). See **RFC 6698** and updates.

The basic idea behind DANE is that a domain owner can make their public TLS keys available on their authoritative DNS nameserver. The same chain of trust that authenticates A records from this nameserver will also authenticate the certificates. That is, certificates are signed by local KSKs, and those in turn are signed by parent KSKs, until the root zone is reached. The long list of certificate-authority keys stored in browsers would be reduced to the much shorter list of root-zone DNSSEC keys.

certificates are stored as DNS objects of type TLSA. Certificates are specific to applications, so the domain name (*eg* `www.example.com`) is prefixed with the port and transport protocol, each with a leading underscore "_" (443 is the port number for https traffic):

```
_443._tcp.www.example.com
```

DANE solves a number of problems. First, it makes (or potentially makes) expensive certificate authorities unnecessary. Second, certificate authorities have to verify that someone asking for a certificate for a given domain is actually the owner of that domain; under DANE, this is solved by having the domain's authoritative nameserver be responsible for both the domain and the keys.

Finally, DANE simplifies the process of adding new certificates, and makes this pretty much a do-it-yourself operation. Once the owners of a domain have created a zone-signing key and had it signed in the parent-zone DS record, they can create as many additional certificates as they want. The validation library used by clients (whether incorporated into the TLS library or not) would be able to ensure that certificates so issued are for the domain in question, or a subdomain, and also verify the port and protocol constraints above. Each such DANE certificate would be verifiable through the DNSSEC chain of trust. More-flexible certificate creation would eliminate the need for the "private" (self-signed) certificates often used to validate access to VPNs and email servers. Under DANE, certificates could become lightweight objects created on a moment's notice. (Of course, no browser currently supports any of this.)

While DANE is intended for the creation of TLS (X.509) certificates, other forms of verifiable public keys would also be possible. See **RFC 6394** for further examples.

While compromise of an authoritative nameserver using DANE could lead to false TLS certificates, this can also happen with conventional certificate authorites, who generally use control of the domain as evidence that the entity requesting the certificate actually owns that domain.

Compromise of the `.com` nameserver would allow the distribution of false DANE-based TLS certificates for any `.com` domain. But if that happens we are all in trouble, regardless of DANE.

There is one subtle difference, which is that of trust. Under the conventional certificate-authority regime, uses can *in principle* decide which CAs they trust. While this is rarely exercised by browser users, it is available as a potential option. Under DANE, to trust the certificate for `example.com` is to trust the `.com` and root nameservers; there is no choice.

For some, the fact that the `.com` and root nameservers are under control of the United States government (through IANA and agreements with IANA) is of concern. It is easy enough for other governments to create their own hierarchy, starting with the two-letter country-code domain and an associated trust-anchor key, but it is still a government in charge. However, while certificate authorities are private entities, it is not clear to what extent they can or would resist government demands to issue misleading certificates.

22.12.3 Why Isn't DNSSEC More Popular?

SSH and TLS have been adopted eagerly by system and website administrators. Even SNMPv3 (*21.15 SNMPv3*) has seen widespread adoption, following early concerns about deployment difficulties. But while many governments (such as the United States) have mandated DNSSEC for their own websites, and while organizations with ties to Internet governance have been early adopters (especially ICANN; see this statement), DNSSEC adoption by major Internet corporations has been vanishingly small. Using the dig tool, it is straightforward to verify that none of the domains in the table below support DNSSEC (none return RRSIG records when A records are requested with the `+dnssec` flag, as of 2019):

apple.com	alibaba.com	amazon.com	ebay.com	expedia.com
facebook.com	google.com	microsoft.com	netflix.com	salesforce.com
tencent.com	twitter.com	uber.com	yahoo.com	youtube.com.

Google has supported DNSSEC at their public resolver, 8.8.8.8, since 2013, but not at their authoritative nameservers. Paypal.com is one of the few large sites that *has* adopted DNSSEC.

These companies are seriously interested in Internet security. Why do they turn their backs on DNSSEC?

The primary answer is that if a site enables DNSSEC, and *some third party* makes the wrong DNSSEC-configuration mistake, then the site becomes unreachable, for at least some users. It goes dark. It is cast into the void.

And there is no error message for those users.

If a TLS certificate is faulty, at least the end user gets a message in their browser; see the sidebar "Certificate Errors" at *22.10.2.1 Certificate Authorities*. The browser directly manages the TLS connection, and, if something goes wrong, the browser learns what, and can inform the user. The majority of certificate errors have to do with misspellings and expirations and updates and other things under control of the domain owner; pretty much the only other major source of TLS errors is the certificate authority, and the domain owner can keep an eye on them.

But with DNS, most browsers farm out lookups to a system resolver library; that library likely does not return anything to the browser that might distinguish DNSSEC validation errors from NXDOMAIN errors. The site or public resolver that the local system sends its DNS queries to likely also does not return different results for validation errors versus NXDOMAIN. And while an error in DNSSEC validation can occur at the root, it can also occur at any lower level in the DNS hierarchy. It is impossible for domain owners to track potential problems at every possible resolver.

A second reason sites choose not to make use of DNSSEC is that it is often perceived as unnecessary. If a user receives a corrupted address through DNS, the problem should be detected through TLS certificate-signing failure. There have been remarkably few DNS-based attacks for which DNSSEC would have made the difference in terms of detection.

A large-scale series of DNS attacks in 2018 involved changing the IP addresses retrieved via DNS for selected hostnames to those of servers under the attackers' control, just as with cache-poisoning attacks (*7.8.3 DNS Cache Poisoning*). These attacks, however, entailed the compromise of the **authoritative nameservers** involved; the attacks were achieved through stolen DNS-administrator credentials (and possibly through misfeatures in the DNS-configuration software within some domain registrars). See this FireEye blog for more details. DNSSEC does *not* protect against an attacker who is able to compromise a zone's authoritative nameserver. Nameserver credentials (all login credentials, really) must be guarded carefully.

The counter argument is that DNSSEC certainly has the potential to block *some* attacks, and that more security is always better than less. This latter concept is called the defense in depth principle. It is, in the abstract, hard to argue with. In the concrete, on the other hand, it depends. DNSSEC is complex, and introduces its own risks. The defense-in-depth principle has been used to argue for regular password changes, but that policy seems to have the paradoxical effect of encouraging users to choose shorter, easier-to-remember passwords. The real question is whether DNSSEC adds *enough* to Internet security to be worth the change.

Eventually, as most public DNS servers and ISP-based site resolvers gain experience with DNSSEC, switching over to it at the server side will become safer and safer. It is likely that, eventually, the going-dark risk will be seen as negligible; at that point, DNSSEC starts to make a lot of sense even if the added benefit is modest.

Another concern is that DNSSEC is old, and the world of DNSSEC still contains a great many legacy 1024-bit RSA keys. By today's standards, those are not considered long enough for long-term use. But DNSSEC does not mandate such short keys. As we saw in an example above, the `.com` key-signing key was 2048 bits; root keys and other top-level KSKs are similar.

There is still the issue that the domain hierarchy is more tightly controlled by governments, but while some individual users may be concerned about this, it is not clear that large Internet companies are.

22.12.4 DNS over HTTPS

An entirely different approach to securing DNS – or at least to reducing DNS problems – has been taken by Mozilla in their Firefox browser. The idea behind DNS over HTTPS – **DoH** – is to abandon the use of site resolvers, and have Firefox send DNS queries to one central DNS resolver, secured by TLS. The specifics are in **RFC 8484**.

One stated goal is to eliminate DNS cache-poisoning attacks by, in essence, switching Firefox users to one large, highly secure resolver (a "trusted recursive resolver" or **TRR**) that gets all its data directly from authoritative nameservers. Another goal is to prevent ISP eavesdropping on (and even interference with) DNS requests sent to local ISP-provided resolvers. Of course, the ISP-eavesdropping concern is simply replaced by the concern that the central DNS resolver is collecting DNS queries.

The TRR partner selected by Mozilla is Cloudflare, manager of a large CDN. There is an agreement negotiated between Mozilla and Cloudflare that personally identifiable data in DNS requests is to be discarded after 24 hours, and never shared with outside parties.

Use of DoH by Firefox is controlled by the `about:config` setting `network.trr.mode`; a value of 0 prefers the normal system resolver and a value of 5 completely disables DoH. One can also set the resolver itself, *eg* to a non-Cloudflare one, though it must be one that understands DNS-over-HTTP queries.

The DoH specification spells out how DNS queries are to be mapped onto HTTP GET requests; traditional DNS queries and responses look nothing like HTTP.

There is a separate approach, DNS over TLS; it is outlined in **RFC 7858**. The goal is simply to prevent eavesdropping on DNS traffic from hosts to their individually chosen resolvers.

22.13 RSA Key Examples

In this section we create a short RSA key, using the `openssl` package, available for Windows, Macs and Linux. We then break it, via factoring.

The first step is to create an RSA key with length 96 bits; this length was chosen for easy factorability.

```
openssl genrsa -out key96.pem 96
```

The resultant file is as follows:

```
-----BEGIN RSA PRIVATE KEY-----
MFICAQACDQCo1hzP6/gTzbNAEHcCAwEAAQINAJzcEHi8aYSO0iizgQIHANkzC28P
jwIHAMb/VQH8mQIHALc+9ZqRyQIHAKkfQ43msQIGJxhmMMOs
-----END RSA PRIVATE KEY-----
```

This is in the so-called PEM format, which means that the two lines in the middle are the base64 encoding of the ASN.1 encoding (*21.12 SNMP and ASN.1 Encoding*) of the actual data. Despite the `PRIVATE KEY` label, the file in fact contains both private and public keys. SSH private keys, typically generated with the `ssh-keygen` command, are also in PEM format.

We can extract the PEM-file data with the next command:

```
openssl rsa -in key96.pem -text
```

The output is the following:

```
Private-Key: (96 bit)
modulus:
    00:a8:d6:1c:cf:eb:f8:13:cd:b3:40:10:77
publicExponent: 65537 (0x10001)
privateExponent:
    00:9c:dc:10:78:bc:69:84:8e:d2:28:b3:81
prime1:
    00:d9:33:0b:6f:0f:8f
prime2:
    00:c6:ff:55:01:fc:99
exponent1:
    00:b7:3e:f5:9a:91:c9
exponent2:
    00:a9:1f:43:8d:e6:b1
coefficient:
    27:18:66:30:c3:ac
```

The default OpenSSL encryption exponent, denoted e in *22.9.1 RSA*, is 65537 = 2^{16}+1. (The default exponent used to be 3, but see exercise 10.0)

We next convert all these hex numbers to decimal; the corresponding notation of *22.9.1 RSA* is in parentheses.

modulus (n)	52252327837124407964427358327
privateExponent (d)	48545702997494592199601992577
prime1 (p)	238813258387343
prime2 (q)	218799945153689
exponent1	201481036403145
exponent2	185951742453425
coefficient	42985747170220

We now verify some arithmetic, using any tool that supports large integers (*eg* python3, used here, or the unix `bc` command). First we check n = pq:

```
>>> 238813258387343 * 218799945153689
52252327837124407964427358327
```

Next we check that ed = 1 mod (p-1)(q-1):

```
>>> e=65537
>>> d=48545702997494592199601992577
>>> p=238813258387343
>>> q=218799945153689
>>> (p-1)*(q-1)
52252327837123950351223817296
>>> e*d % 52252327837123950351223817296
1
```

To encrypt a message m, we *must* use efficient mod-n calculations; here is an implementation of the repeated-squaring algorithm (mentioned above in *22.8.1 Fast Arithmetic*) in python3. (This function is built into python as `pow(x,e,n)`.)

```
def power(x,e,n):  # computes x^e mod n
    pow = 1
    while e>0:
       if e%2 == 1: pow = pow*x % n
       x = x*x % n
       e = e//2    # // denotes integer division
    return pow
```

Let m be the string "Rivest". In hex this is 0x526976657374; in decimal, 90612911403892.

```
>>> m=0x526976657374
>>> c=power(m,e,n)
>>> c
38571433489059199500953769621
>>> power(c,d,n)
90612911403892
```

What about the last three numbers in the PEM file, `exponent1`, `exponent2` and `coefficient`? These are pre-computed values to speed up decryption. Their values are

- `exponent1` = d mod (p-1)
- `exponent2` = d mod (q-1)
- `coefficient` is the solution of coefficient × q = 1 mod p

22.13.1 Breaking the key

Finally, let us break this 96-bit key and decrypt the message with ciphertext c above. The hard part is factoring n; we use the Gnu/Linux `factor` command:

```
> factor 52252327837124407964427358327
52252327837124407964427358327: 218799945153689 238813258387343
```

The factors are indeed the values of p and q, above. Factoring took 2.584 seconds on the author's laptop. Of course, 96-bit RSA keys were never secure; recall that the current recommendation is to use 2048-bit keys.

The Gnu/Linux `factor` command uses Pollard's rho algorithm, and, while serviceable, is not especially well suited to factoring the product of two large primes. The author was able to factor a 200-bit modulus in just over 5 seconds using the msieve program, one of several large-number-factoring programs available on the Internet. Msieve implements a version of the number-field-sieve algorithm mentioned in *22.9.1.2 Factoring RSA Keys*.

We are almost done; we now need to find the decryption key d, knowing e, p-1 and q-1. For this we need an implementation of the extended Euclidean algorithm; the following Python implementation is taken from WikiBooks:

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)
```

A call to `egcd(a,b)` returns a triple (g,x,y) where g is the greatest common divisor of a and b, and x and y are solutions to g = ax + by. From *22.9.1 RSA*, we need d to be positive and to satisfy 1 = de + (p-1)(q-1)y. The x value (the second value) returned by `egcd(e, (p-1)*(q-1))` satisfies the second part, but it may be negative in which case we need to add (p-1)(q-1) to get a positive value which is congruent mod (p-1)(q-1). This x value is -3706624839629358151621824719; after adding (p-1)(q-1) we get d=48545702997494592199601992577.

This is the value of d we started with. If c is the ciphertext, we now calculate m = pow(c,d,n) as before, yielding m=90612911403892, or, in hex, 52:69:76:65:73:74, "Rivest".

22.14 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0 Modify the netsploit.c program of *22.2.2.3 The exploit* so that the NOPslide is about 1024 bytes long, and the NOPslide and shellcode are now *above* the overwritten return address in memory.

2.0 Disable ASLR on a Linux system by writing the appropriate value to /proc/sys/kernel/randomize_va_space. Now get the netsploit attack to work *without* having the attacked server print out a stack address. You *are* allowed beforehand to run a test copy of the server that prints its address.

3.0 Below are a set of four possible TCP-segment reassembly rules. These rules (and more) appear in *[SP03]*. Recall that, once values for all positions j⩽i are received, these values are acknowleged and released to the application; there must be some earlier "hole" for segments to be subject to reassembly at all.

1. Always accept the first value received for each position.

2. Always accept the last value received for each position, up until the value is released to the application.
3. [“BSD”] For each new segment, discard each position from that segment if that position was supplied by an earlier packet with a left edge less than or equal to that of the new packet. Values in any remaining positions replace any previously received values.
4. [“Linux”] Same as the previous rule, except that we only discard positions that were supplied by an earlier packet with a left edge *strictly smaller* than that of the new packet.

For each of the following two sets of packet arrivals, give the reassembled packet. Values in the same column have the same position. Reassembled packets will all begin with “daa” and “ebb” respectively.

(a).

```
| |a a a a a
| |    b b b b b b b
| |    c c c c
|d|
```

(b).

```
| |    a a a a a
| |b b b b b
| |    c c c c c c
| |      d d d
|e|
```

4.0 Suppose a network intrusion-detection system is 10 hops from the attacker, but the target is 11 hops, behind one more router R. Outline a strategy of tweaking the TTL field so that the NIDS receives TCP stream “helpful” and the target receives “harmful”. Segments should either be disjoint or cover exactly the same range of bytes; you may assume that the target accepts the first segment for any given range of bytes.

5.0 Suppose Alice encrypts blocks P1, P2 and P3 using CBC mode (*22.7.3 Cipher Modes*). The initialization vector is C0. The encrypt and decrypt operations are E(P) = C and D(C) = P. We have

- C1 = E(C0 XOR P1)
- C2 = E(C1 XOR P2)
- C3 = E(C2 XOR P3)

Suppose Mallory intercepts C2 in transit, and replaces it with C2’ = C2 XOR M; C1 and C3 are transmitted normally; that is, Bob receives [C1’,C2’,C3’] where C1’ = C1 and C3’ = C3. Bob now attempts to decrypt; C1’ decrypts normally to P1 while C2’ decrypts to gibberish.

Show that C3’ decrypts to P3 XOR M. (This is sometimes used in attacks to flip a specific bit or byte, in cases where earlier gibberish does not matter.)

6.0 Suppose Alice uses a block-based stream cipher (*22.7.5 Block-cipher-based stream ciphers*); block i of the keystream is Ki and Ci = Ki XOR Pi. Alice sends C1, C2 and C3 as in the previous exercise, and Mallory again replaces C2 by C2 XOR M. What are the three plaintext blocks Bob deciphers?

7.0 Show that if p and q are primes with p = 2q + 1, then g is a primitive root mod p if $g \neq 1$, $g^2 \neq 1$, and $g^q \neq 1$. (This exercise requires familiarity with modular arithmetic and primitive roots.)

8.0 Suppose we have a *short* message m, *eg* a bank PIN number. Alice wants to send message M to Bob that, without revealing m immediately, can be used later to verify that Alice knew m at the time M was sent. During this later verification, Alice may reveal m itself.

(a). Suppose Alice simply sends M = hash(m). Explain how Bob can quickly recover m.

(b). How can Alice construct M using a secure-hash function, avoiding the problem of (a)? Hint: as part of the later verification, Alice can supply additional information to Bob.

9.0 In the example of *22.7.7 Wi-Fi WEP Encryption Failure*, suppose the IV is $\langle 4,-1,5 \rangle$ and the first two bytes of the key are $\langle 10,20 \rangle$. What is the first keystream byte `S[ S[1] + S[S[1]] ]`?

10.0 Suppose Alice uses encryption exponent e=3 to three friends, Bob, Charlie and Deborah, with respective encryption moduli n_B, n_C and n_D, all of which are relatively prime. Alice sends message m to each, encrypted as

- $C_B = m^3 \bmod n_B$
- $C_C = m^3 \bmod n_C$
- $C_D = m^3 \bmod n_D$

If Mallory intercepts all three encrypted messages, explain how he can efficiently decrypt m. Hint: the Chinese Remainder Theorem implies that Mallory can find $C < n_B n_C n_D$ such that

- $C = C_B \bmod n_B$
- $C = C_C \bmod n_C$
- $C = C_D \bmod n_D$

(One simple way to avoid this risk is for Alice to include a timestamp and the recipient's name in each message, ensuring that she never sends exactly the same message twice.)

11.0 Repeat the key-creation of *22.13 RSA Key Examples* using a 110-bit key. Extract the modulus from the key file, convert it to decimal, and attempt to factor it. Can you do the factoring in under a minute?

12.0 Below are a series of public RSA keys and encrypted messages; the encrypted message is c and the modulus is n=pq. In each case, find the original message, using the methods of *22.13.1 Breaking the key*; you will have to factor n and then find d. For some keys, the Gnu/Linux `factor` command will be sufficient; for the larger keys consider msieve or some other fast factorer.

Each number below is in decimal. The encryption exponent e is always 65537; the encryption is c = power(message,e,n). Each message is an ASCII string; that is, after the numeric message is converted to a string, the byte values are each in the range 32-127. The following Python function may be useful in converting numeric messages to strings:

```
def int2ascii(n):
    if n==0: return ""
    return int2ascii(n // 256) + chr(n % 256)
```

(a) [64 bits] c=13467824835325843134

    n=15733922878520524621

(b) [96 bits] c=8007751471156136764029275727

n=57644199986835279860947893727

(c) [104 bits] c=6642328489179330732282037747645

n=17058317327334907783258193953123

(d) [127 bits] c=95651949760509273124353927897611145475

n=122096824047754908887766043915630626757

Limit for Gnu/Linux `factor` without the GMP library

(e) [185 bits] c=14898070767615435522751082309577192810119252877170446296

n=36881105206579952723396854897336450502002018818436622611

(f) [210 bits] c=1030865591241775131254813948981782525587720826169501849049177362

n=1089313781487492651628882855744766776820791642308868127824231021

(g) [280 bits]

c=961792929180423930042975913300802531765775361218349440433358416557620430721970697783

n=1265365011260907658483984328995361695181000868013851878620685943648084220336740539017

(h) [304 bits]

c=17860252858059565448950681133486824440185752167054796213786228492658586864179401029486173539

n=26294146550372428669569992924076340150116542153388301312743129088600749884420889043685502979