MALWARE
Motivation behind malware (spyware etc) today is crime, i.e. to be able to steal information in some way that can be later used for monetary gains. This has given rise to a healthy black market for trading exploits where people make money in this way.

The initial few slides are mostly introductory. Please refer to them for more information, apart from what is captured here.

Goal of the software is spam, extortion, phishing and the likes.

The forms that malware take: viruses, worms, botnets, rootkits and spyware.

Virus:
- Replicates itself
- Historically the term has been used for malware that can't function as a standalone entity. It needs a host to do its job (e.g. pieces of software, hard disk – when the OS boots up, the virus gets loaded into memory and is then free to do whatever it wants to). However those types of viruses are not common anymore. The main idea although remains that the virus should get into the system (loaded in memory and running) before a lot of other things (which could prevent the virus) can. The idea behind rootkits is similar.

Macro viruses: Attach themselves to things like Microsoft Documents.

Types of viruses that are most common today are those that are spread with email attachments and so on.

Side Discussion: Exploits vs. Viruses
Traditionally exploits have been thought of as something which is short lived. It might lead way to something bigger e.g. a virus (say by means of downloading something). Exploits don't have a persistent footprint (they are small and simple) and are entry mechanisms, whereas viruses are persistent, and they stay around for a longer time.

Worm:
They are closely related to viruses, the technical difference being the fact that worms can function as standalone entities. They don't need any other host program to function. One of the oldest known worms was the Internet worm of 1988 (Morris worm). After a hiatus, around 2002-03 there were a few which were reported. The purpose of the later ones was to primarily cause DOS attacks and show off technical prowess. However the novelty wore off, and apart from the inconvenience to users caused by such large scale attacks, these attacks were not helping the attackers in any way. More recently the worms have been stealthier, they spread slowly to avoid detection. e.g. Code Red, Slammer.

The more recent worms are scanning worms. i.e. once the worm starts running, it looks for other places where it could spread, and so it is scanning for vulnerable hosts by sending exploits and observing what happens. e.g. on a LAN, it could be looking for other machines on the same LAN and so on. Once a fair number of hosts were affected, the network would be so overloaded with these scanning kind of activities, that apart from the general slowdown in the network, the propagation of the worm itself
would also be affected.

One of the more recent worms has been the Storm worm which is still making people wonder about its functioning. It establishes botnets and uses rootkit-like technologies for obfuscation and to hide its behavior and presence.

**Botnets:**

These are typically collections of compromised computers. The idea being that an attacker would make use of this botnet in order to do things like spam and DDOS attacks. He gets a footprint on many machines, which is required because an effective attack would typically require a lot of bandwidth and CPU power. And secondly there is the question of detection and eventual shutoff of compromised nodes. So there a larger no. of compromised computers would help. And finally with so many nodes, tracing the attack to the attacker is difficult.

Typically for capturing sensitive information, one doesn't need to have a botnet.. A spyware should be enough.

**Rootkits:**

The origin of the word is from the fact that a rootkit is something that would allow someone to subvert root and be able to have root privileges without other components having the knowledge that the rootkit is there. The primary purpose of a rootkit is to maintain invisibility. So there is some piece of malicious code running on the system which wants to remain invisible. The malicious code might be doing various things.. like running processes, creating/deleting files etc.. The purpose of the rootkit is to hide all of these things.

The early versions of the rootkits which came out in the 90's were user-level rootkits, meaning that the rootkit itself consisted of some user level programs. The focus was on typical programs that a sysadmin would use to get system state e.g. ls. So what the rootkit would do is to replace the standard versions of such programs with newer versions which would filter out the rootkit's own data while doing a directory listing. Similar things could be done with ps (listing out processes on the system), netstat (if the malicious code is running a server, then the modified netstat would not list that), login (for providing some sort of a backdoor entry, e.g. a special login name which allows passwordless login etc.) So by replacing a reasonable no. of these user-level programs earlier rootkits were able to hide themselves. But then the approach was not fool-proof, because a custom program which used system calls to figure out system state, would then see those hidden rootkits.

Hence came the need for kernel-level rootkits. The most obvious thing to do is system-call interception because if there is a custom program to examine system state, at the end it needs to make system calls to gather system information. So if the malware controls the system call itself, then it can control everything. An analogy can be drawn as to how an operation like listing of a directory can be controlled by malware by changing system call logic, and similarly for listing of processes and network connections etc. A rootkit can hence hide in the system call table and sanitize necessary information so that the user never sees its presence. That is part of the reason why Microsoft at some point decide that it should not expose the system call table and also the Linux kernel developers, because it made it too easy for these rootkits to hide themselves.
Side discussion: Rootkits vs. Viruses
The key characteristic of viruses is that they spread. A rootkit is an additional defense that a collection of malware might use to hide themselves.

However things have become much more complex now. Say the rootkit is sitting at the system call level and sanitizing information. Now there could be a custom program which opens /dev/mem or /dev/kmem and monitors kernel memory. Sanitizing such information coming from /dev/mem and /dev/kmem such that the rootkit remains hidden is extremely difficult due to the multitude of ways of examining system state. So for the rootkit to be undetectable by any of these mechanisms gets quite difficult. What is easier is to do things which were done at the user-level earlier, and to hide that type of information. As the tools for rootkit detection become more sophisticated, it becomes more and more tougher for rootkits to remain obscure.

e.g. one of the things that people started doing was having some kind of checksum on kernel code. At this point it gets tough for the rootkit. So what the rootkits tried then was to hide inside of data structures, the idea being that its code is there is kernel memory, but at places where it isn't obvious that its part of code. There are lots of data structures in the kernel memory which have code pointers. So by changing these code pointers the root could stick itself in the middle. e.g. an interrupt handler. In this case there aren't any kernel threads running on behalf of the rootkit, which makes it difficult to figure out that there is a piece of malicious code.

In such scenarios where there is an attacker and a defender, the attacker always has the upper hand in devising ways to prevail. The defender could stop all malicious entities from entering into the system of course, but in that case the system usability goes down considerably.

One of the most recent threads is virtual machine based rootkits, where what happens is that the rootkit goes underneath the OS and becomes a virtual machine. So when one looks around in one's own OS (or what he think is his machine), there is no record of anything because the rootkit is hiding underneath the OS. How does the rootkit go under the OS: If the rootkit has complete access to the system, then it can change things that happen at boot time etc.

As a sideline, its unrealistic to say that a rootkit can't be detected by any means at all! What is important here is that a typical user under typical circumstances would not be able to detect it. Also a rootkit would need to have root privileges to get installed. On the UNIX side, they are often in the form of system software updates, or things that one would do with root privileges. On Windows installations are anyways to be done as Administrators. Windows Vista does have some form of integrity levels, wherein things which one downloads from the internet say, can't just do anything on the system without user intervention.

SonyBMG DRM Rootkit (2005) kicked up a storm. They would use a rootkit like technique to monitor all the processes that accessed the Music CD on the system and wanted to prevent anything other than Sony's Music Player from accessing the CD (essentially to stop people from copying and distributing music illegally). The main problem was that it allowed other malware to piggyback on top of this capability.

Spyware:
Rootkit can be thought of as an infrastructure for malware to operate, since malware's primary goal is to do something else rather than just hide itself which is a secondary goal. Spyware can be built on top of a rootkit. The main goal of spyware is to steal sensitive information like passwords. These days the focus is not solely on ads. Spyware would rather be there on a system on a low profile, not disturbing users to the extent that they would remove it immediately.

The way spywares work is that they intercept keystrokes (they register themselves to be notified on every keystroke), and forward it to the process that is supposed to get it. Or snoop on network interfaces to see what data is being sent on the network. Another form of spyware common on Windows is in the form of browser plugins. One advantage of being at a higher level than keyboard logging is that the spyware then knows the context (can see higher level events) and can figure out just what it needs to intercept. For example, the password field in a form that a user is filling up.

Apart from this the spyware might just monitor what the user does, e.g. browsing activities and then send spam matching that.

**Spam :**

People were gaining some control over spam in 2006, but now its almost out of control. Around 90% of the total email traffic is spam! This is where botnets come handy. If the source of spam is limited, then it becomes relatively easy for users to just block a small set of addresses. So a lot of PCs on the internet are used as email forwarders making it tough to blacklist every one of them. The spam sites keep changing pretty soon.

**Phishing :**

Phishing is masquerading as a trustworthy person. The goal is to defraud somebody. Talking of websites, once the initial request goes to the attacker's website then they can do anything. Either serve copies of the original websites or download it on demand and serve it out to the user (probably a combination – caching of part of the original website).

**Technical aspects of malware :**

Malware wants to remain stealth. For reasons like..

- It doesn't want to get detected and be thrown out of the system
- The malware writer would want to protect his own IP. For every new malware, once its functioning gets detected, someone (for instance companies like Symantec) would come out with defenses for it. That would mean that malware writer would have to figure out new ways of doing things. If the malware is difficult to detect, then that would fetch them a higher price.

**Detection of viruses :** Historically viruses were detected by their signatures on disk by virus scanners. These days the scanners can do stuff online. i.e. for example when network data is coming in, the scanner can figure out from byte sequences in the packets if the packet is infected with a virus, and then block it. So the primary method is content examination and matching it against a sequence of bytes (signature) that is supposed to be unique for the virus.

**Polymorphic viruses :**
In order to defeat this virus writers started relying on mechanisms like *polymorphism*. They were using some kind of an encoding technique e.g. encryption. So by changing a key, you change how your payload looks and therefore a detection mechanism which looks for a specific sequence of bytes would no longer work. When the virus replicates, it will replicate its body with a different encryption key, making the body different.

This caused quite some trouble for antivirus writers till they figured out what was happening. There are only a limited set of tools for generating polymorphic viruses. So they started focusing on those tools and their characteristics to be able to figure out signatures. In addition they also started augmenting static scanning techniques with dynamic scanning techniques which come into picture when the virus is beginning to run.

The first technique which focuses on the tool figures out what kind of encryption/decryption routine (which is in plain text) the particular tool is using. The terms packing/unpacking are more often used than encryption/decryption in this field.

The other technique is run-time detection. So when the code is unpacked and starts to run, that can serve as a trigger for virus detection. Based on these 2 techniques, the antivirus writers were able to tackle polymorphic viruses that exist today. Although the exact way in which they examine the code is not necessarily public, but it might be done inside a virtual machine like environment.

**Metamorphic viruses**:

Firstly there isn't any particular reason why the polymorphic and metamorphic viruses are called so. For metamorphic viruses, there is no particular encryption/decryption going on. Instead there is code that morphs itself using techniques like replacing instruction sequences with other instruction sequences that have the same meaning. This however is not relatively easy and requires run-time dynamic transformation. One would need a disassembler, need to capture the semantics of instructions so that it can be figured out that a set of instructions can be replaced with a different set. The tools for doing this are hence hard to develop.

The advantage is that there is absolutely no fixed part anymore. The code is in place text, just that instruction sequences are being replaced. To this day there aren't any good solutions against metamorphic viruses.

To protect IP, malware these days is intelligent enough to figure out if they are running inside of a simulation environment like a virtual machine. And if they can do so, then they don't do anything bad so as to avoid being caught. There are techniques to detect a virtual environment. e.g. presence of certain files. In practice its difficult to hide the presence of it. Another detection technique is to time the difference between different computations. e.g. its a known fact that inside a virtual machine I/O intensive operations run much slower whereas CPU intensive operations run at almost the same rate as on the host machine. So a loop of a million times vs. I/O operations can give an indication of the operating environment. Most malware comes with some level of anti-virtualization defense built in.

**Program obfuscation**:

Program obfuscation techniques are being used in 2 domains..
• Intellectual Property (IP) Protection for legitimate processes
• Intellectual Property (IP) Protection for malware

These techniques can be used to achieve the following..

• Make it hard to disassemble code: a) One could use encryption so that disassembling is ruled out. But the problem with this is that someone could run it inside of a virtual environment, and wait for the trigger when it is decrypted, and then start analysis. There are tools which continuously encrypt. e.g. decryption is done, the code runs and then gets encrypted again, which means that there is not enough time when the code is out in the open. This of course has performance overheads but again that can be tolerated in certain situations especially if the encryption technique is not heavy-weight.

   b) Other ways of making it difficult to disassemble code are to insert data in the middle of code, violating typical ABI conventions (doing only jumps instead of calls, or the return from calls is offset by a few bytes from where it should ideally be, or jump to the middle of code etc.). With all this in place it becomes very difficult to figure out the control flow. Typical disassemblers make a few assumptions like return from calls at fixed places, for conditional expressions both branches are taken at some point in time.

• Higher level obfuscation techniques: These techniques are not only relevant for binary code, but are also useful for HLLs such as C or Java code. In this technique a statement is broken into pieces which are dispersed across the code with jumps, thereby destroying the code structure totally. To make it a little more difficult to be tracked, one should use conditional jumps with conditions which are always TRUE or FALSE. This makes analysis tough. This is called Opaque Predicate, whereby just a look at the predicate is not enough to figure out if the predicate is TRUE or FALSE.

   Similarly by inserting junk code guarded by conditions that never hold TRUE, using alternate code sequences and aliasing one could achieve program obfuscation.