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.

PROGRAM OBFUSCATION

PROGRAM OBFUSCATION:

In the previous lecture we had briefly been told about Program Obfuscation. We take a deeper look at Obfuscation and its techniques.

<u>INTRODUCTION:</u>

What is Program Obfuscation?

Program Obfuscation means taking a program and 'modifying' it so that the modified program has the same behavior (giving us the same results) as the original program did. But its code otherwise bears a different resemblance to the original code.

Looking different means that, any "adversary" who has access to the program code or the binary should not be able to figure out the internal working of the program.

** The reason for Program Obfuscation is to prevent or make hard the process of reverse engineering.

Why would someone want to reengineer code?

- Security analysts observe the behavior of a software to see if it performs malicious activities or not. If the software under analysis is indeed a malware, the malware author would desire that its analysis be difficult or the code be convoluted enough to make the "adversary" believe it to be non-malicious.
- Sometimes security analysts observe behavior of malware to understand its behavior and control flow. In such cases the authors of such malware desire that the analysis not reveal the correct functioning of the code.
- Adversary software companies wouldn't mind if they could 'understand' the functioning of software made by a competitor and build better software or sell the same software by a different name.

> By analysis we mean the following:
> A binary has been disassembled and the disassembled code is under analysis to get an understanding of the functioning of the program. OR
> A static analysis of the program code is being performed to get an understanding of the functioning of the program.
> Program Obfuscation can be thus performed at various levels.

Games are another example where the creators of the game wouldn't want someone to gain an unfair advantage by modifying the code.

Java Scripts are being deployed widely to be run locally on machines through web browsers these days by proprietary software developers. Developers would obfuscate the source code to make it hard for competitors to understand the behavior.

But if someone spends good enough time on such obfuscated code, it won't be long before he/she figures out the behavior of the target program. So obfuscation is not a fool proof way to realize

Intellectual Property Protection.

Now, obfuscation is not only restricted to help malware developers to help prevent analyzing their malware code as noted above.
More general applications do exist.

GENERAL OBFUSCATION APPLICATIONS

- Intellectual Property Protection
  -There are proprietary algorithms developed by developers as a part of an application which they would not like competitors to gain access to. They would like the code to be as intangible as possible.
- Watermarking
  -Embed some kind of "mark" in proprietary digital data or code. IF "something" (code or data) is stolen (or used in an undesirable way) this could be figured out.
  -How this works is the fact that the stolen data will be useless as the "mark" and data are tied together in a way that the data is useful only if the "mark" exists in a way it is supposed to.
  -It is also called Steganography.
  -If an "adversary" figures how and where this "mark" is stored and more importantly the connection between the data and the mark, he could remove the watermark and use the code/data the way he wants.
- Tamper Resistance
  -Say you buy an e-book and Acrobat sells software to read it. Acrobat not only needs to make sure that it has some mechanism to ensure that the e-book is not copied and distributed but also that it is read only using the software it sold.
  -If acrobat sells a dedicated device in which such an e-book is stored, it is easier for Acrobat to ensure that there is no way you get this data out and use it.
  -But if it is software, then it becomes harder. Acrobat will 'encode' data in some way that only the "special software" can read it. Maybe along with the data, there is some information which specifies the display it uses, and the user who is intended to use it and so on. Non-encoded (or with no such other information with it), the data is useless. If the "adversary" can figure out the association between and the data and the meta-information, maybe he can modify it and use it the way he wants.

Technically speaking, for all the above applications, if an "adversary" spends enough time on the application, he can subvert the protection mechanisms. IF the data uses any kind of encryption, then the key to decrypt the data will be stored inside the software. Figuring out what the key is then is just a matter of time, thus defeating the whole mechanism.

What are Proprietary software developers doing about it?

Developers are developing technologies that make the "subverting" difficult enough that it is simply not worth trying to subvert. If an adversary is unable to figure out what encryption algorithm is used where the key is stored and in what format then the obfuscation technique is indeed worth using.

Ways to make it tough for subversion of obfuscation?

Maybe the key for encryption is a combination of hardware-device-identifiers of some devices on the system and so on. The adversary will have a tough time figuring out which devices they are combined with figuring out the subset of the identifiers which are being used to make the key. These could be ways in which vendors/authors make it hard to subvert obfuscation mechanisms.

We now take a look at various Obfuscation techniques.

<u>OBFUSCATION TECHNIQUES</u>

Program obfuscation is a general research area. Research has been going on for the last 10 years. Initially nobody realized the need for it.
General perception: Binary code is hard to figure out by itself.
This perception changed after Java came in. Java byte codes were found easy to reverse engineer which forced people to start thinking of ways to make the process of reverse engineering (understanding the functionality of binaries) harder. Another motivation was copyright protection of music and videos. (DRM is an outcome of this motivation)

Types of Obfuscation Techniques:

- Control Flow Obfuscation
- Data Obfuscation

Obfuscation against Static Analysis

Obfuscation Techniques

Obfuscation against Dynamic Analysis

Control Flow Obfuscation
-moving code around to make code intangible

Data Obfuscation
- making it hard to understand the changes data undergoes.

*** Control Flow Obfuscation (CFO): ***

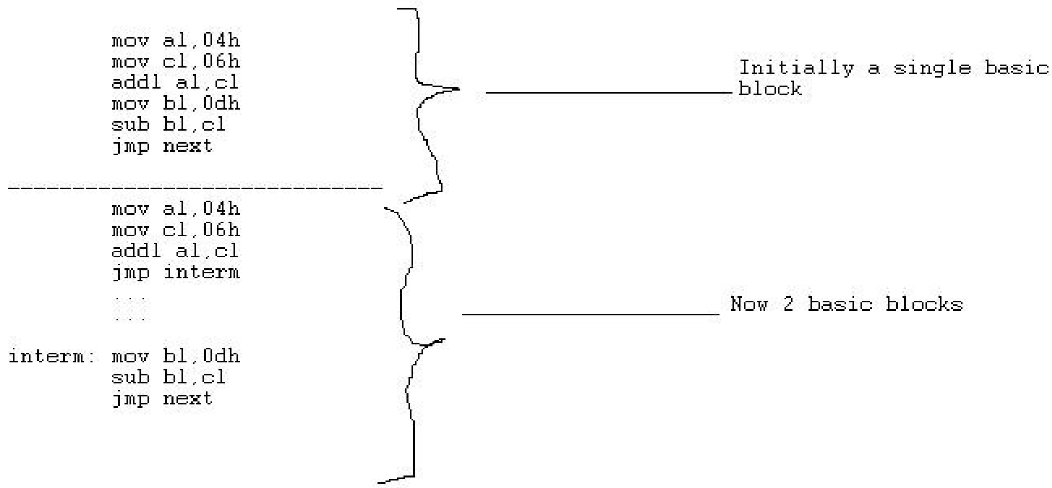
Changing the control flow of the program can be termed as Control Flow Obfuscation.

<u>Renaming Functions</u> is a type of CFO. This may have no relevance in some form of executables. When binaries are made, they are stripped of all function names and symbol information. This works well at a higher level.

Obfuscation can also involve <u>splitting or merging blocks of code</u>. A Basic block can be broken into multiple basic blocks.

> Basic block is a sequence of instructions that has one entry point (the first instruction of the block) and one point of exit (the last instruction of the block) with no branching or jump instructions in the middle.

Splitting basic blocks can involve something like this:



The basic blocks can then be re ordered any which way (as long as the functionality of the program is preserved).

<u>Merging/Splitting Loops</u>: The same can be done with loops also.
--Say, a single loop with 2 statements could be turned into 2 loops with one statement each. This can be difficult but it does somewhat make code intangible to the adversary. It is always possible to achieve something like this. The two new statements will now be dependent on each other in some way.
--Similarly, multiple loops can be merged into one single loop.

The idea again is to make code as intangible as possible.

<u>Merging/Splitting Procedures</u>: Just like loops the same can be done with procedures.
A single procedure can be broken into multiple procedures.
Say, a procedure A can be broken into procedures B and C. Thus (when A is called) B will be called followed by a call to C.
Merging of 2 or more procedures into one procedure is possible.
Calling 2 procedures when only one is to be called is another possibility. There could be cases when calling an extra procedure C when only Procedure B is called will not hurt the functionality of the program in any way.

Obfuscation can be done at any level. All techniques are applicable at all levels. But there are some methods of obfuscation which are possible at a lower level which are not possible at higher levels.

Obfuscation at Lower Level vs Obfuscation at Higher Level
At a lower level, you could eliminate all procedures, have no CALLs and instead have only JMPs. Then during disassembly analysis becomes difficult with observer seeing RETs with no CALLs.

Procedures could be either inlined or outlined.
-- Inlining a procedure means that instead of calling a procedure, the procedure is taken and physically inserted at the point where the CALL to the procedure exists.
-- Outlining means that a sequence of instructions which could be grouped as performing functionality, could be taken a made into a procedure and a CALL or a JMP is inserted instead.

Once this kind of splitting or combining is performed it could be reordered to achieve further obfuscation.

Add DEAD CODE in the original code. Dead code means unreachable code i.e. the set of code will never be executed but is there to make the code intangible. Of course there is the need to make it NOT obvious that a particular code is dead code. For instance by putting such code after a JMP would make it obvious that it will never be executed. Desirable thing would be to insert conditions to execute the dead code and also making sure that these conditions are never satisfied so the dead code is not executed.

--Defeating human analysis, defeating techniques which take low level instructions and combine them to give a higher level view of what is going on are the main challenges.
--An important technique which is needed to be avoided is pattern matching which observes a sequence of low level instructions and decides that the pattern translates to some higher level program structure.

> An important thing to note is these obfuscation techniques should be automatic. Questions might arise as to where then these obfuscation techniques are implemented. If it is done at the higher level, then the compiler might optimize them! So it makes sense that these techniques be integrated with the compiler at the right place. Some obfuscation techniques (Data Obfuscation) could actually be done at a higher level as the compiler is unlikely to figure out this and optimize thereafter.

Each of the individual techniques mentioned above are simple as a standalone technique. The real advantage would be when they are combined and then applied iteratively in any particular order. This would be something close to true obfuscation.

*****Data Obfuscation: (DO) *****

This involves obfuscating data instead of control flow.

What is the motivation?
An adversary could also try to figure out the behavior of the program by looking at the transitions that variables/data goes through. The idea is to make it difficult for the adversary to understand these transitions.

Techniques are:
Renaming variables:
This bears no significance at the lower level as data at lower levels are identified and referenced by registers and memory location addresses

Splitting/Aggregating variables:

- Taking individual variables and putting them into a structure or an array. Arrays and structures denote uniformity or represent something as a whole. By putting unrelated variables into a structure or an array an adversary might misinterpret them to be something else.

- 2 variables represent a single variable. Say variable A is now represented by variables B and C as B-C. When B and C are kept separately, it becomes harder to realize that they represent one single variable.

What about obfuscated data interacting with unobfuscated code?

Global variables are not shared between modules. It is rarely found. Say Shared Objects (SOs). Routines are shared and not global variables. Routine names are exported or imported. Internal data representation is the local affair of a program.

<u>Moving variables:</u> from static area to heap or local variables so that it is harder to differentiate local variables from global variables and dynamically allocated variables.

<u>Other techniques:</u>

--Cloning variables
--Inserting junk elements into arrays.
--Encrypting data values.

Decrypt the data when it is needed to be used and then encrypt again. OR use some encryption algorithm or encoding mechanisms that are property preserving. (Order preserving encoding would involve adding a constant offset to a variable and performing arithmetic on it without subtracting the offset.)

--Introducing additional levels of indirection (ALIASING).

Instead of a static variable we have a pointer variable which is initialized. Aliasing could be introduced. Aliasing is a hard analysis problem.

--Introducing memory errors:

Accessing single variables as array members.

Say:

```
…
int x[10]; -------- > (1)
int y;
int z;
z = z +1; ------- -> (2)
…
```

Statement 2 is easy to understand.
A memory error here could be referencing z as x[11].
(2) can now be written as:

```
x[11] = x[11] + 1;
```

Statements like these are memory errors BUT if permitted, clearly make it confusing for an adversary to analyze the transitions a variable goes through. Moreover if someone is working with binaries it is difficult to figure out the extent/bounds of arrays.

--Adding/Removing Function parameters

Useless parameters could be added and maybe the function performs a lot of operations on these useless parameters. But the actual operation could only have a few lines of code.

As can be noted, the above techniques assume that an adversary has access to the binary (which can be disassembled) or the source code itself.
So we can categorize these techniques as **"Obfuscation against Static Analysis"**

Let us look at obfuscation from a different perspective.

Obfuscation against Dynamic Analysis

The Obfuscation Techniques which we've seen above would most probably work well against static analysis.
However it would not work well against dynamic analysis.

Say, that a program is run in a virtual environment and the adversary observes the system calls which are made by the program. This is the only understanding of the program which an adversary gets in such case. Obfuscation wouldn't help in such analysis.

But, a dynamic analysis does have a limited scope as far as understanding algorithms goes (by simply looking at its inputs and outputs).

<u>Anti virtualization techniques</u> were discussed in brief last class. These techniques are employed by and large by Malware which try and detect if they are being tested/analyzed in a virtual environment. (One such way is by running many I/O operations and non I/O operations and then observing their ratio.)

I/O operations are really slow in VM environment as compared to a non-VM environment, while CPU operation performances are almost the same.

What is the motivation behind Dynamic Analysis?

Intuitive feeling is that Static Analysis would be a real tough task especially since such analyses are automatic and the mechanism can definitely be subverted.
These analyses techniques are merely defenses/solutions developed after newer and newer Obfuscation Techniques were discovered. These analyses may not even completely detect all obfuscation as newer & more complex techniques are developed.
A better compliment would be to see how a software interacts with the System concerned by monitoring System Calls.

Such analyses are performed in a Virtual Environment generally. But Malware, as mentioned could detect this and avoid exhibiting its malicious behavior or it may not be that devious as in, the subject could be a BOT waiting for a command from its BOT MASTER.

A good thing to do could be to combine Static and Dynamic analysis. When unable to figure out what is happening by Dynamic Analysis, use Static Analysis.
<u>Example:</u>
--In Dynamic Analysis, you see that certain paths are not being exercised but want to see the dynamic behavior of those paths. You might need to change a variable value for that. It could however be that there is another variable that needs to be changed too! (…according to the logic of the program). If not

then something else could happen (crash maybe).
--This correlation can be figured by means of static analysis. Thus we see the combined use of Static and Dynamic Analysis.
--Limited Static Analysis with Dynamic techniques is thus a good combination.

<u>Summarizing Malware</u>:

- There is plenty of motivation for attackers to remain stealthy. They use techniques like Obfuscation and Anti-virtualization.
- Malware (and their writers) are adaptive. It exhibits evasive techniques to circumvent checks and analysis to achieve its objective and not be discovered.
- Note that, as mentioned above, the analysis techniques are "generally" built after realizing that there are certain peculiar behaviors exhibited by Malware and so on.
  Thus these are reactive mechanisms, whereas Malware behavior could be termed as active.
- Of course if there is a unique detection mechanism which has been developed which is unknown to a large population maybe it will detect & analyze the Malware behavior.
- There is a difference between defenses against Exploits and defenses against Malware. For Malware the need to assume a very strong adversary model is desirable.
  - In exploit detection, it is assumed that program being exploited is a trusted program. Even after plugging in defenses in the program, the program does not actively try and subvert the defense mechanism.
  - A Malware on the other hand tries actively to subvert any defense mechanisms setup against it, to achieve its objective.

- The defense mechanism setup against the Malware should be self protecting.
  If the Malware finds out a way (loophole or a vulnerability in the defense) to disable or subvert the defense mechanisms, then it doesn't make sense.
- This of course is hard to achieve, owing to various factors including System Design, underlying OS and the Architecture. People may be forced to run certain programs with certain privileges. There could be a Malware in it which installs a Rootkit in the system which makes things tougher for the analysts.
- Complete Mediation is another issue.
  Checking if Malware does only 'x', 'y' or 'z', the Malware could do task 'a' and evade the mechanism.
  Thus every action taken by the Malware should be detected and out under the scanner.
- Multi step Attacks (Stepping Stones) is another characteristic of Malware.
  Malware may not achieve its objective in one step, but rather perform a series of steps which leads to its objective in a not so obvious manner.
  This is one of the classic techniques for detection evasion. Behavior based detection techniques look for certain behavioral patterns and block it if found.

Defense against Malware is a current problem being faced by all. It is definitely a topic of intense research.

<u>Summarizing other WORRIES:</u>

Insider Attacks:

- One class of attacks is the kind when somebody from outside tries to exploit a vulnerability in a system.

- Another class would be owing to social engineering, the attacker gets a piece of code running into a system, making it now vulnerable
- A third class would be when there are people inside who are “malicious”. This of course is a major issue in Financial Institutions in the form of Insider Tradings.

  People are dealing with Insider Attacks through Policies and Law Enforcements.
- A way of achieving this would be to first detect one. This is achieved by means of extensive logging and auditing or access control mechanisms.
- Access control systems tend to be rigid, so they are uncommon to see.
- The reason behind avoiding access control is : There could be exceptions which need to bypass an access control.
- Insider attacks differ from organization to organization.
- Say for a Software developing Organization, one of its developers embeds vulnerabilities in the code. Potential for damage increases manifold as the same software is used in many (thousands of) computers
- It becomes hard to pin point and moreover prove criminal intent. Law Enforcement as a deterrent works little.
- This is an instance of Cyber Warfare or Cyber Terrorism.

This concludes the topic of Malware and Obfuscation techniques.