Main:

Introduction:

Hello.

This talk will cover various areas involved with making binaries harder to analyse, reverse engineer and modify. Additionally it will cover various mechanisms that can be used to watermark a binary.

Accompanied with this presentation is a relatively in-depth paper on the subject as well, mainly covering implementation of such techniques on the Linux operating system. The slides and paper will be on the recon cd. Additionally, the talk notes and slides will be available via the felinemenace papers section.

In the interest of having this presentation applicable and useful for everybody, I've decided to keep it at a moderate technical level, and for some areas this covers, include some basics on the subject. However, striking a good combination between the two is difficult.

If you have questions during the presentation, I'd prefer it if it was brought up at the end of the talk. Thanks.

Defence in depth:

Defence in depth is a term borrowed from the military and it is used to describe defensive measures that reinforce each other and that hide the defenders activities from being observed.

Defence in depth usually starts off by defining the problems, rating their significance, and then analysing the problems in depth. Once a clear picture has been obtained, potential methods and ideas to protect the system is then evaluated.

Clearly identifying what you're trying to protect against is important as its easy to go off and lose track of your end goals.

As time goes by, you'll need to re-evaluate what you've implemented, how you've implemented it, and what other people have done with regards to your approach. They could of found methods that bypass its protection altogether, or significantly weakened it. If this is the case, you'll need to examine what they've done, and what methods you can use to make your protection better and thwart their attack.

Defence in depth (cont):

By analysing your plans for an attackers point of view, you will be be able to identify what you would of proposed to do as being weak, or not worthwhile to implement due to time constraints involved. Ask yourself what you can do things to make it more robust, or that attackers have less chance of modifying the protection binary successfully?

During the attack analysis, you will probably think of additional methods that will help your efforts in thwarting an attacker. These should be carefully analysed to see if they are effective, and if its feasible to implement.

If you haven't analysed things from other peoples perspective before, it can take a little bit of effort to get used to. Learning what the attackers are capable of, and testing the things out yourself, will help improve your plans for preventing them.

One aspect which some people seem to sprout off as being a good idea, is to somehow damage ones computer if they detect software that can be used to analyse a programs operation, such as softice.

While the motivation aspect of this can be seen from a developers eyes, there are many cases where people will legitimately have such software on their machine, for example, driver developers, or security analysts. As there are tools out there to hide various debuggers from being seen by applications, it usually isn't that much of an added protection. Therefore, you'll run the risk of annoying legitimate customers of your software.

In additional to that, in many countries it is illegal to do deliberately damage the computer systems. Given those points, it would be a bad idea to do anything that would break various laws in the countries where your software is sold.

A better, and more constructive use of your time would be to work out various ways of misleading and slowing down a potential attacker on your software.

For example, If you detect a debugger actively debugging your binary, you could change various algorithms so that it results in incorrect execution, or invalid values being used. Be sure to separate cause and effect so they don't recognise immediately that the logic choice you're using is going to cause it to break.

By utilising various techniques to inhibit an attacker, you'll increase the amount of time, motivation and skill needed to break the protection schemes around the binary, and thus provide more time until, eventually, the attackers goal has been reached.

Defence in depth (cont)

To summarise the slide, in general, a majority protection schemes generally implement various layer defences in the sense that you can attack and remove layers generally without impacting the operation of the protected binary.

If a protection scheme is going to be robust against analysis and attack, it needs to have it so that attacks on the protection overall reach into the core stability, usability and functionality of the program.

I'd like to end this section off with a quote from Nietzsche, "He who fights with monsters should look to it that he himself does not become a monster. And when you gaze long into an abyss, the abyss also gazes into you.". The relevance of this is left up to the people present to determine.

Watermarking

<watermarking history>

Watermarking can be useful in many situations, such as tracking and identifying whose copy of the software was released to various warez groups for example. By letting your users know you have the capability to identify who released the software, you deter them from doing so.

Apart from the deterrence factor, another benefit from watermarking is obvious if you have the ability to either prosecute the people responsible, or, the ability to sully the reputation of the company who is responsible. Unfortunately, watermarking doesn't prevent against general fraud to obtain the software.

In general, a fragile watermark is used for authentication and integrity attestation, and is used to guarantee that an item, for example, a document had came from the right place. An attack on these type of watermark would be to modify the item without breaking the watermark.

On the other hand, we have robust watermarks. Robust watermarks are generally used for copyright purposes, or authenticating who had originally inserted the watermark. An attack on robust watermarks would to modify the item in some way to either render the watermark ineffective, or to remove the watermarks presence.

Watermarks are also have another property which indicates if the watermark is visible or invisible. Visible watermarks are meant to be publically recognisable, and can be human perceivable. Invisible watermarks on the other hand are meant to be recognisable only to authorised people, and thus makes it harder to know where to attack to break the watermark.

Depending on how much effort you wish to put into designing the watermarking system, and how its going to be used, there are various methods that can be used to insert the watermark.

The most simple one to insert is a basic incrementing counter. This system is obviously weak if an attacker has more than one copy of the software. When an attacker has multiple copies of differently watermarked items, analysing it is known as a conclusive attack, as the attacker has multiple copies available to analyse.

Other areas that can be used for storing the watermark in can be the order of certain operations. An often cited example is how various initial values for a function are loaded. For example, 2 variables can be initialised in two different orders.

This method is also vulnerable to the above described attack. The best method to work around this is to make each binary as different as possible. For example, the compiler flags, such as optimisation flag, can be changed, the order of which object files are read in can be modified, the layout of structures can be randomised per compile for each customer. In turn, the transformation of the binary by the randomisation can be used to

identify whose copy it is.

There has been some research into the feasibility of making software watermarks tamper proof so that it becomes a lot more difficult to remove watermarks without impacting the correct operation of the program.

To summarise a research paper entitled (XXX), they make a graph and parse it during run-time to obtain data constants used in the program, and use the obtained value in an operation, such as addition, or printing a string.

Because the data used for returns are also used in parsing the graph, in theory, if you change the value, the correctness of the program is in jeopardy in other parts of the code that rely on it.

Obfuscation

Obfuscation is the act of deliberately making something harder to analyse and understand.

The aim of obfuscation is to increase the time, skill, and cost required to analyse the protected code. Obfuscation can be applied at both the source code level, and assembly level.

The probably most famous obfuscated source code would be that from the International Obfuscated C Code Contest (also known as IOCCC), which is a annual competition that has been running since 1985, and is where people can submit their obfuscated code to be judged.

While obfuscated source code may be useful for some purposes, it is not the end product that is shipped to the end customers, and thus generally isn't the best place to focus your efforts.

Some people have previously tried to obfuscate stuff by having large amounts of junk code, which doesn't really help for obvious reasons.

Some viruses, such as JunkComp, used techniques such as:

- Prefixes on instructions that it doesn't make sense on. For example, repeat while not zero on a subtraction instruction.
- Opaque conditionals to trick control flow analysis
- Alias opcodes (for example, mov eax, ebx can be encoded two ways)

While those effects measures were semi-effective, the mechanisms were being used as a wrapper around the true virus code. Since the true virus code could be reached, it could be analysed and taken apart.

Implementing a serious obfuscation engine is a lot more involved, and has various aspects and things to consider.

Code layout obfuscation covers the area of changing the layout of the code. For example, it might be splitting a function into small blocks and spreading them all over the binary, changing variable lifetimes and variable ordering.

Data obfuscation covers making references to data harder to analyse. For example, it could be converting static data to into a function where the string is generated, inserting more references to data to make it harder to locate a certain object you're after.

Control obfuscation covers breaking inserting opaque conditionals, making logic choices harder to determine by static analysis, duplicating blocks of code and independently obfuscating each block, and making changes to the control flow. Some of those changes could include flattening the control flow so that its not irreducible.

Pre-emptive obfuscation covers inserting specific obfuscation constructs to target a

particular problem. Some things that are worth targeting are how hard it is to automatically unobfuscate a binary, and how hard it is for a human to unobfuscate it.

Obfuscation (cont)

As determined by Christian Collberg, Clark Thomborson and Douglas Low in their paper entitled “A Taxonomy of obfuscating transformations”, obfuscation is classified by its potency, resiliency, and cost. For more information, see the above mentioned paper.

The term potency refers to how much time, effort and skill is required by a human to understand, and remove, the obfuscation construct. Various methods may dramatically increase the potency of the algorithm, but may not necessarily reflect upon the other targets.

The term resiliency refers to how much time, effort and skill an attacker needs to write a program to automatically unobfuscate a construct, and how much resources the unobfuscator requires to run. Increasing the resiliency will help prevent automated unobfuscators.

The term cost refers to about the impact of implementing the previous two methods on the execution of the program. For example, a method that causes huge memory usage or a long execution time to run would be said to have a large cost.

By using these metrics, it is possible to objectively rate the usefulness of particular constructs and decide if and when to include them.

Obfuscation (cost)

Control flow obfuscation

Opaque constructs are code blocks which always result in the same output. The general aim of an opaque construct is to be difficult to identify and remove, thus increasing the resiliency of the affected code.

Opaque constructs can be inserted and used in logic decisions and data decisions.

The term “instruction context” refers to analysing a block of code, and determining if the instructions depend on a certain order of execution, what ways the registers are used and determining the lifetime of the registers.

Once that information has been gathered, it is possible to rewrite the instruction context. Some possible things that might be done are modify which registers are being used, what arguments are being passed to instructions, and insert irrelevant instructions.

By sampling the instructions in the program that you are obfuscating, you can make it so that the instructions you're inserting are similar to the ones that already exist, and in the same frequency as previously. This helps hide where the inserted instructions were placed.

In order to make automated analysis harder of the inserted instructions, serious thought should be given to make it so that the program reads and writes to memory after operating on registers.

This then makes it significantly harder to automatically analyse as the resulting code, as the instructions can be no longer marked as spurious if they're shown that the registers are stored to, and then just stored into again, without being read from.

Automated analysis of control flow loops can be made harder by inserting new constraints inside loops. The aim here it to break standard compiler-generated loops so that it is harder to automatically identify and recover them.

This could be via an opaque construct which always ends up adding 0 to a value which is used in the conditional check, inserting spurious instructions between initialising values and adding extra comparisons which has no effect.

Data obfuscation

Data obfuscation is where you obscure the data used in the program and make it harder to find where various pieces of data are instantiated and used.

By converting static data into pieces of code, you can make it so that it is harder to analyse and modify, as people need to first locate where the data is being initialised, and what constraints are being placed on it

The references to static data can be easily obtained in the object files during compilation time, as there will be relocation entries which patch up the binary during the linking stage. This allows us to easily convert pieces of data into functions without worrying about following execution flow, and finding all references to it.

Obfuscation (cont)

By adding new cross references to existing data blocks, it makes it look as if that data block is used more than what it appears to be, and additionally slightly more memory for in the analysis program to hold all the cross references.

By modifying the vtable in c++ classes, the functions inside a class can be rearranged, and new junk functions can be added or inserted into the class. If the class is to have its vtable rearranged, the obfuscating code would have to track through all the appropriate code, and modify them so they point to the correct entry.

As well as modifying a class's vtable, you could modify structure layouts and add extra items of various types and preform dummy operations on it. This increases the time involved with reconstructing what a specific structure is used for, and what its members are, and how they are referenced.

In order to increase the time needed to analyse structure usage, you could implement a simple linked list or doubly linked list that gets modified over time.

In order to increase time involved with analysing parts of a binary, certain data types could be converted into a class. The class would then be responsible for performing the various operations on the file, such as addition, and subtraction. Additionally to that, it could be made to appear that the class is also doing a lot more.

Control Flow obfuscation

Code can be split into so called basic blocks by identifying blocks of instructions that don't rely on previous instructions to perform their operations. Once the chunk of code that is to be optimised has been broken down into basic blocks, they can then be operated on.

If you have some instructions inside a basic block that don't rely on each other, the instructions could be swapped around to make it slightly harder for humans to analyse.

Additionally, after the basic blocks has been gathered, you can independently obfuscate them. Some of the operations that you might add to the the blocks are add reversible operations to the basic blocks, such as adding / subtracting a number to a register, or changing which register is being used for certain operations.

The operations will need to be reversible so that if there is a basic block that has other code references to it, they can converge and be continue executing fine.

Code flow reduction is where you remove modify the code flow conditionals to be flat as possible, thus removing a fair amount of information available about a program's behaviour, and depending on implementation, a fair bit harder to recover that information.

Disadvantages to obfuscation?

The control flow reduction could be achieved by implementing a switch table, and setting up the switch value inside the code, or other methods such as calculating the address to jump to, and jumping to it from a central point. The more resistant it is to static and runtime analysis, the better the method is, and the harder it will be for people to recover the code flow.

In general, you'll want to reserve obfuscation for the important functions, as it will have an adverse performance impact. The effort of implementing a decent obfuscation algorithm will also impact upon delivery times for the software.

Of course, you could go for a commercial product to implement them, but you won't necessarily know how good they are, and people who are interested in attacking your software will have a slight advantage as more people will have analysed that protection scheme for implementation weaknesses.

Licensing schemes (change?)

A decent licensing implementation will take a fair amount of effort to implement so that it is stable, reliable, and provides a decent amount of security against determined attackers. Ideally, you'd like the licensing scheme to be robust even after they've paid for the software.

It seems obvious, but if you don't want someone to have access to pieces of code or data in your demo version, don't include it in the demo version. If the license scheme is based upon on people entering a license code, you could generate a decryption key from the license data. Depending on the implementation, you may accidentally make it so that the protected data or code can be trivially recovered.

In order to improve the robustness of the licensing code, the licensing code should directly related to the correct operation of the program, thus causing incorrect operation should an invalid license code be entered.

One method of doing this is to utilise the license code in logic and data choices in your code. With logic choices, you would have two versions of code, but one would be slightly buggy. With data choices, you could use it to set flags correctly for functions that takes flags. You would have to make it hard to identify where the checks are coming from by making it hard to follow the data.

The aim behind this is to make it extremely hard to follow all the logic choices in the application, and to follow all the data paths as well. This will result in people having to spend significant time analysing the code and data flow trying to determine if its relevant.

If the license code is based upon people entering a string, you can make life a little bit easier for them by including a tiny 1 or 2 byte checksum. This allows you to inform the user if they've entered an incorrect license code without giving anything away about the validity of the license code, or how it was generated.

This also provides a slight detour for an attacker, as they may see the checksum compare and think they need to patch their, or alternatively, they'll try and write a license key generator.

By using the license code information without checking it for correctness, you are avoiding giving away potentially critical hints to an attacker, which in turn means they may have to spend more time analysing the code to see what it is doing.

After sketching out your license scheme, how it operates, and what it will affect, think about ways you can break the system. Depending on what you're aiming for, you may want to improve upon what you have already outlined, and find other methods.

Virtual Machines

Virtual machines are self-contained, emulated machines which have their own assembly language, and memory areas. They can be extremely simple, or extremely complex. Such a complex one would be Java, although the Java implementation has a lot to be desired with how hard it is to analyse.

Virtual machines can be complete byte code driven, or they can use a Just In Time compiler to generate code that is executable on the native CPU the virtual machine is running on.

Since Virtual Machine instructions and implementation can be whatever the creator dreams up, they usually involve a fair amount of analysis to determine what opcodes are available, and what they can do. Then they have to get familiar with how the code implemented in the Virtual Machine is put together and what constructs are used, which makes it harder and more time consuming to analyse.

The general disadvantage with Virtual Machines is that they take a fair amount of effort to implement and design, and then to write the code that is executed in them. Additionally, generally, they only need to be analysed and understood once, before they use loose their usefulness.

This can be offset however, because its possible to randomise how the byte sequence is decoded, what the bytes map to, and how parameters to the op code is accessed. This allows for a bit of leeway. Additionally, some other measures can be taken, for example, the byte sequence to be decoded can configure various parameters of the virtual machine.

Bastardising the file format 1

The general idea behind the modification of the executable file headers is to cause the program using to analyse the file to misbehave in some way, such as crashing the program or system, and generating incorrect output from what would be expected of the program being used, and in general to make the process a bit more painful for people.

Some techniques involve stripping the section tables or munging the symbol table from ELF files, inserting incorrect program headers which the analysis tools parse, or specifying invalid sizes which cause various tools to loop for a while and allocate large amounts of space.

For ELF, a lot of tools use the section table, and thus its more useful to make that completely misleading. A method of doing this is to append another binary to your executable, and pointing the section header to that ones section header, and fixing up the file offsets so they all point to the other binary.

New methods can be developed by analysing how the operating system and associated tools load the binary, and comparing with how the analysis tools parse the binary, and using the differences to attack the tools.

However, this is a standard arms race. Once you use these methods, or develop new methods, people will identify the cause of the problem, and will come up with fixes, even if it involves hand-patching the vulnerable binary.

Bastardising the file format 2

There are a couple of various disadvantages with this, which may merit concern. For starters, there is portability between OS's and various emulators, such as WINE. Most people tend to expect they can use their programs across different releases of OS's, and some people like to be able run their software on other operating systems under emulators such as WINE.

Additionally, occasionally you may like to debug your own binaries, and you can't reconstruct the errors when you use the binaries that haven't been modified. And sometimes when you modify binaries in certain ways, anti-virus software picks them up as being suspicious, which then reflects badly upon your software.

Summary

In order to effectively slow down and deter attackers from breaking what you've done, the best method is to utilise multiple layers that interlock, and that aim to protect each other from being analysed. The more robust and reliable the implementation is, the longer it will last.

- don't check values for consistency / correctness, just use them straight away
- learn to attack your own implementations, in order to identify weaknesses
- Perhaps keep an eye out on forums / etc
- Realise when and where to focus your efforts.
- Have fun in the process.

Summary (cont)

reword, make longer. perhaps mention trusted computing.

However, if people are significantly determined, nothing is unbreakable given todays open hardware. While there is trusted computing in the horizon, time will tell if it is sufficent