Summary reverse engineering

Group: The Chess Masters

Introduction

In this report we will summarize the content described in the following papers.

- https://hgi.rub.de/media/emma/veroeffentlichungen/2012/03/27/itit.2012.0664.pdf
- http://www.cs.ucsb.edu/~vigna/publications/2004_kruegel_robertson_valeur_vigna_USENIX04.pdf
- https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_CASED/Publikationen/2010/KinderVeith08a.pdf
- https://www.cs.arizona.edu/solar/papers/CCS2003.pdf
- http://www.cs.ucsb.edu/~chris/research/doc/virology06_dynamic.pdf
- http://vxheaven.org/lib/pdf/Static%20Analysis%20of%20Binary%20Code%20to%20Isolate%20Malicious%20Behaviors.pdf

This report will contain a summary for each article and will be closed with a conclusion.

Reverse Code Engineering – State of the Art and Countermeasures

Binary Reverse Engineering refers to the process of trying to understand the the components and relationships of a binary executable. This analysis is often performed by trying to reverse binary code to a higher level language.

Today, there are an increasing number of totally legitimate applications of RE, such as closed source auditing, vulnerability research and the analysis of malware.

The main fields of program analysis are control flow analysis (how is code reached and where does it lead to), data flow analysis (the relationships between data structures), and type recovery.

Control Flow Graphs (CFGs) are used to construct diagrams showing the sequential execution of code. When program analysis problems need to be examined from a graph theoretical viewpoint, it is possible to make use of Program Dependence Graphs.

When trying to reverse engineer binaries there are several difficulties which may arise:

- Compilation of binaries may result in information loss. This is due to elimination of dead code, no comments, compiler optimizations etc. The compiler optimization results in a code which is different than the original.
- Hardware knowledge is necessary to understand the code.
- Programs may contain anti reverse engineering routines.

In general there are two methods of reverse engineering: static analysis and dynamic analysis.

Static analysis

Static analysis is preceded by a process called disassembly, which generates assembly code from the binaries. After disassembly, the assembly code can be decompiled to a high level source code. Hardware related features are removed and high level data structures are recovered. A successful decompilation will not result in the same code as the programmer had originally written, due the the problem described above. Adding to those, another significant problem is the lack of runtime memory.

Dynamic Analysis

Dynamic analysis executes the actual binary and is often done with a debugger. Powerful debuggers are OllyDbg, WinDbg, gdb and Immunity Debugger. A debugger places an additional layer between the OS and the program; this allows for the insertion of breakpoints and thereby the ability to pause and resume the program at the debugger's convenience. A debugger can also make use of something called tracing, with which it is possible to trace back a call to a certain function or instruction. Tracing can happen on multiple levels of granularity.

For many types of programs behavioural analysis is very important. When working on a real machine, opposed to working on a virtual machine, it possible to make use of a technique called hooking. This allows hooked function to be redirected to a user supplied target, so that the function can be analysed. When working in virtual machine, a technique called Virtual Machine Introspection allows monitoring of a VM from outside.

Countermeasures to Reverse Engineering

Passive Protection methods

Some of these passive methods are already discussed, for example code optimization. This results in code obfuscation which can make it hard to analyse code semantics and data structures. By using correct obfuscation techniques it is possible to conceal the flow of the program and thus make it impossible to find the targets of certain functions or indirect branches. Other methods to obfuscate code are by using opaque predicates (non trivial boolean expressions), exception handlers (makes it hard to know the active handler) and multithreading. A programmer can also choose to destroy the input table, so that the debugger cannot resolve which system services will be used. Another last method is the use of runtime packers, which compress and or encrypt the executable binaries.

Active Protection Methods

Anti Debugging
A program can query its environment and analyse the environment in which it is being executed to possibly find out whether it is being run in a debugger. Consequently is may choose the crash the debugger or try something else to avoid being analysed. Another way to prevent a binary from being spawned in a debugger is by attaching an own debugger to itself. A process can be debugged by one debugger at the time, and so the debugger of the analist won't work.

Anti Emulation
Some binaries are run initially in an emulation environment and thereafter they get analysed. However in an emulated environment certain processes or items are missing, such as the Instruction Set. Anti-Emulation protection make use of this incompleteness and e.g. stop execution.

Anti Dumping
Some binaries have multiple layers of encryption and compression. During execution, after the layers have been unfolded the Original Point of Entry is called, during which a memory dump occurs. A method to counter this is by destroying parts of the binary file after making an encrypted copy. The binary is later reconstructed in dynamically allocated memory, making the dump file invalid, since the dynamic memory regions are ignored.

Static Disassembly of Obfuscated Binaries

This article contains a new technique for performing static analysis on binary files, so that the disassembly process has more success. This is useful because some disassemblers fail when the binary file is obfuscated. Obfuscation is of course the process of making a binary file more robust against reverse engineering.

Introduction

In the first section they explain that software reverse engineering tools are used to translate the binary file to a higher level representation, (partly) automatic. The difference between disassembly and decompilation, is that a disassembler tries to convert bit patterns to machine instructions(assembly) and that a decompiler tries to convert the constructions to a higher level language. Reverse engineering is made more difficult by obfuscation techniques. This article focuses on disassembly phase and in particular the techniques described by Linn and Debray.

Obfuscation makes it harder to steal ip from benign companies, it also gives attackers the opportunity to hide malicious code.

There are two types of disassembly techniques, dynamic and static. Static analysis, given a binary input, outputs the assembly instructions by inspecting the binary input. Dynamic analysis parses the instructions based on several executions of a binary file. The disadvantage

of dynamic analysis is that it does not always cover all instructions since it is possible that not all instructions in the binary file were execute, dynamic analysis is execution dependent. Static analysis has the disadvantage that it might take junk code into account. In the article they only use static analysis.

There are two main approaches for static analysis:

1) Linear sweep, which scans the binary file from beginning to end in a linear fashion. It is prone to errors because one wrong translations might cause a chain of errors.
2) Recursive traversal, traverses the program based on its control flow. This has the advantage that it does not parse unreachable code. It has the disadvantage that, if the target of a control transfer instruction cannot be determined, the disassembler will fail to reconstruct the program flow. To make up for this speculative disassembly(linear sweep) is used for the parts which could not be reached.

Targeted obfuscation techniques

The authors of this article developed a disassembly method which is more robust against certain obfuscations. They take care of the obfuscations presented in the article by Linn and Debray.

Linn and Debray use two new obfuscation techniques:

1) inserting junk bytes. These are junk bytes are inserted at locations which cannot be reached during run time. A linear sweep however will try to parse the bytes into instruction. So a linear sweep will fail. It will not affect the recursive traversal however because it follows the control flow, so it will not reach unreachable code.
2) The second technique is a branch function. A branch function basically obfuscates function call. All function calls go through the branch function and it executes the function. The branch function adds an offset to the return address, so the branch function does not return to instruction following the call instruction. So an offset number of bytes are unreachable, here junk bytes can be inserted and since all static analysis techniques assume that the address after branch is the address to return to, they try to parse junk bytes, causing an incorrect disassembly.

Assumptions

Disassemblers are based on (compiler)assumptions, if one of these assumptions is violated the disassembler does not work anymore. The obfuscation by Linn and Debray violates the assumption of state-of-the-art disassemblers, so a new disassembler with other assumptions is presented. The disassembler uses the following assumptions:

1) No overlapping instructions.
2) Conditional jumps do not point to junk bytes. If a obfuscator can do this the disassembler does not work anymore.
3) Junk bytes can be inserted at unreachable locations.
4) Control flow does not continue after call. So after a call junk bytes can follow.

Algorithm

The algorithm presented makes use of two techniques, general and specific. General does not require knowledge about the obfuscator, specific does. The algorithm works in the following way:

1) First the binary code is split up into functions, which can be analyzed independently. The functions are detected based on the prolog (specific combinations for making a new frame on the stack).
2) In each function the control flow is analyzed. The CFG(control flow graph) is constructed for each function, which is a graph which shows the flow between basic blocks(pieces of instructions executed in order without jumps). The traditional recursive method is to scan the binary linearly until a jump occurs, then recursively do the same thing for all targets. This does not work for obfuscated binaries because some of the jumps might point to junk bytes. The authors have a different method for creating the graph. They first create an initial graph, taking into account all possible intra-function control transfer instructions (conditional jump in the same function). Note that not all of these instructions are valid, but the valid instructions are a subset of this.
3) After the initial CFG is created, block conflict resolution is applied. Which is removing blocks from the graph which create conflicts within the graph. A conflict occurs when two blocks contain overlapping instructions. Block conflict resolution is done in several steps. First it is stated that each block which is reachable from a known valid block is valid, assumed is that the block containing the first instruction is valid, so each edge from that node is valid. Secondly, if two nodes have a conflict and are both reachable from a third one, the third one can be removed. There are also three heuristics which are used to remove blocks from the graph. This step leads to a reduced CFG.
4) The last step is gap completion and it is used for filling in the parts between blocks with reasonable instructions. The main reasons why these blocks were not evaluated is because it contains either junk bytes, or the CFG algorithm made a wrong decision.

There are also tool-specific techniques which use knowledge about the obfuscation techniques. In the article techniques are described to counter the branch function described by Linn and Debray.

Results

For getting the results, the same programs as in the article of Linn and Debray are used and the same data are used, with exception that the results of the new disassembler are added. The results are measured in disassembler accuracy, which is defined as one minus the difference between the real instruction set and the found instruction set, divided by the real instruction set. The new disassembler has a higher score and the difference between disassemblers is significant. Noteworthy is that even without tool-specific rules the disassembler is very accurate.

Jakstab: A Static Analysis Platform for Binaries

Static analysis is already discussed in the first section. One tool which provides for such analyses is JAKSTAB. JAKSTAB translates assembly instructions to an RTL style intermediate representation through an iterative process and then creates a CFG iteratively from this representation. Indirect branch targets are resolved using dataflow analysis. CFGs created by IDA Pro are often incomplete and can miss calls to imported functions. In the intermediate representation many status flags are not used but overwritten. JAKSTAB thus performs a live variable analysis to eliminate any dead code. All parts of the CFG which are known are constantly being propagated and folded.

Obfuscation of Executable Code to Improve Resistance to Static Disassembly

The article is about making the disassembly of binary files harder. They only consider static disassembly. The reason for making disassembly of programs harder, is to prevent people from inspecting programs so they can find vulnerabilities or can find intellectual property of a company. Note that this is exactly the opposite to the other article, which argues that with disassembly you can analyze malicious software and thus get knowledge about how hackers work. They present some technologies to make disassembly harder.

Introduction

In the article they talk about two types of disassemblers, linear sweep and recursive traversal. They have created methods for decreasing the success rate of these types of disassemblers. One property of disassemblers is that they synchronize pretty quickly after an error, so if you want to thwart the disassembly part you have to maximize the period between the error and synchronization.

Obfuscation methods

All their methods work because of junk bytes. The basic idea is that once the disassembler interprets junk bytes as assembly instructions, they get a sequence of wrong instructions. The junk bytes inserted should not be interpreted during runtime, so you must ensure that the program itself never reaches them. On the other hand you have to trick the disassembler to interpret the junk bytes.

The first method for inserting junk bytes is especially targeted against the linear sweep algorithm. It works by inserting junk bytes before so-called candidate blocks. A block is a sequence of instructions without jumps and executed in order. A block is a candidate block if

junk bytes can be inserted before it. The technique is that after an unconditional jump, junk bytes are inserted. Because the jump goes to a higher address, the intermediate addresses are not used, but they are inspected by a linear sweep algorithm. They convert some conditional jumps into unconditional ones, to make more candidate block and thus decreasing the success rate for the disassembler.

The technique does not work for recursive traversal, because it skips unreachable code. To fool recursive traversal algorithms a branch function is introduced. All unconditional jumps are replaced by a call to the branch function. The function than does the jump to the target (it maps the the address to the targets of the unconditional jump). The branch function does also add an offset to the return address, which gives the opportunity to insert junk after the call instruction. Branch functions do not only obscure the flow of the program but also to create opportunities to mislead the disassembler. Their method works with a perfect hash function in the following way:

1) Apply hash function to return address.
2) Use the hash to find the offset from the return address to the target.
3) Add offset to return address and return.

So an unconditional jump is translated in a function call. The disassembler thinks the next instruction after the call is a valid instruction, whilst it is not. So the recursive traversal disassembler is also tricked.

Results

Results show that it becomes harder to disassemble binaries. It was tested with three disassemblers: IDA Pro (state of the art), objdump(linear sweep) and a custom made recursive traversal disassembler. The results are measured in a confusion factor, which is normalized value of the difference between the real set of instructions and the set of instruction found by the disassembler.

Dynamic Analysis of Malicious Code

Malware experts may get upto 100 samples a day for analyses and therefore it is important that malware analysis can be automated. However, there are several problems to automating malware analysis. Malware can detect if it is being run in a virtual environment, if there are breakpoints and also the analysis environment does not monitor calls to the system, which allows for the malware to evade analysis. The authors present TTAnalyze, which automate malware analysis under dynamic analysis.

There are some inherent weaknesses to static analysis. Firstly, an attacker may deliberately program a malware in such a way that it makes use of obfuscation techniques and thus intervenes with the disassembly process.

Secondly, next to obfuscation the attacker can write the code in such a way that it is hard to analyse the control flow of the malware.

Lastly, the original code may not be the same as the one being decompiled because the programmer made use of polymorphic and metamorphic techniques.

The advantage to dynamic analysis is then that it is immune to obfuscation. The question then remains as to where to execute the binary. A popular option is to run it in a virtual machine, with the only drawback being that malware can detect it is being run in a virtual machine and may then decide to act differently. For this reason TTAnalyze makes use of an emulated environment, which emulates software wise all the processes and hardware of the a pc. It is much harder for a program to detect whether it is being run in an emulator. TTAnalyze emulates an entire operating system running Windows XP instead of just the processor because this increases the chances of not being detected. For this it makes use of a modified version of Qemu.

A malware may make use of Windows API calls, e.g. to confuse a virus scanner into thinking it is not malicious, and native API calls. TTAnalyze must thus (1) be able to track which instructions come from the malware and (2) monitor the operating systems service calls. The first problem is solved by making use of a CR3 register, which keeps track of the address of the page directory of each process. The second problem is solved by comparing the current value of the instruction pointer to the start addresses of all operating system functions being under surveillance.

In the setting created by the above it is not yet clear, if the program executes a function, which arguments are given. This is solved by giving the analist the option to invoke a callback routine, which resolves the arguments. In an emulated system however, it is not always possible to read from the virtual address space. In that case the test subject is forced to read from the page fault handler, which loads the appropriate memory page into the emulated physical memory.

The forcing of these read instructions is done by injecting code into the instruction stream. Because code is run in an emulator, it is easy to insert additional instructions.

Lastly, TTAnalyze provides a way for generating a report containing, general information, file activity, registry activity, service activity, process activity and network activity.

Static Analysis of Binary Code to Isolate Malicious Behaviors

Many products and software nowadays are based on commercial-off-the-shelf (COTS) parts. This could mean that malicious code fragments may be within COTS products. It is possible to use static slicing on binary executables to detect malicious code fragments. The methodology is as follows. First the binary is disassembled and subject it to flow analysis, then it is transformed to a high level language representation, and lastly by making use of slicing techniques, malicious code fragments are detected.

After disassembly, sequences of instructions which perform a logical instruction are grouped together, to so decrease complexity. Then dataflow analysis is performed to make the

code more analysable. Also, parameters and return values can be resolved by using API and library subroutine prototypes. In the paper, Bergeral et al. describe a slicing algorithm which makes it possible to extract essential parts of the code which is assumed to be malicious.

Conclusion

The articles contained similar terminology and covered a wide range of topics within reverse engineering. Some articles focus on the disassembly part, whilst others focus on the decompilation part. The same goes for static and dynamic analysis. An interesting fact is that the intentions of the authors can be opposite to each other. One article contains techniques for making reverse engineering harder, whilst another argues for making it easier. For example one article focusses on making disassembly harder whilst other authors try to make it easier. Both have reasonable arguments for doing so. From this we can conclude that there is a difficult question about the ethics of reverse engineering.

We think that we have a more complete view about reverse engineering after reading these articles.