Edith Cowan University
Research Online

Australian Digital Forensics Conference | Conferences, Symposia and Campus Events

3-12-2007

Forensic analysis avoidance techniques of malware

Murray Brand
Edith Cowan University

Follow this and additional works at: https://ro.ecu.edu.au/adf

 Part of the Computer Sciences Commons

DOI: 10.4225/75/57ad403c7ff2a
5th Australian Digital Forensics Conference, Edith Cowan University, Perth Western Australia, December 3rd 2007.
This Conference Proceeding is posted at Research Online.
https://ro.ecu.edu.au/adf/6

Forensic Analysis Avoidance Techniques of Malware

Murray Brand
School of Computer and Information Science, Edith Cowan University,
Bradford Street, Mt Lawley, Western Australia 6050
mbrand0@student.ecu.edu.au

Abstract

Anti-forensic techniques are increasingly being used by malware writers to avoid detection and analysis of their malicious code. Penalties for writing malware could include termination of employment, fines or even, imprisonment. Malware writers are motivated not to get caught and are actively using subversive techniques to avoid forensic analysis. Techniques employed include obfuscation, anti-disassembly, encrypted and compressed data, data destruction and anti-debugging. Automated detection and classification work is progressing in this field. This includes analysing statistical structures such as assembly instructions, system calls, system dependence graphs and classification through machine learning.

Keywords

Forensics, Anti-Forensics, Reverse Engineering, Analysis Avoidance, Malware Classification Analysis.

INTRODUCTION

Forensic analysis of software that is determined to be malicious could result in the termination and/or prosecution of the author and/or the user of the code who knows its true malicious purpose. This could include logic bombs, viruses, worms, backdoors and trojans. There is no doubt that forensic analysis of software takes time and money. It is to the advantage of the malware author, or deliberate user, that the investigation takes longer than management is prepared to spend on the investigation. Conceivably, the malware author or user would like the malicious component to go completely undetected, and avoid the risk of prosecution.

Scenarios could include programmers writing salami attack style code to assist embezzlement through altering billing algorithms to redirect funds to their own accounts. Programmers could include a backdoor into software so that they can perform some malicious action at a time of their own choosing. Covert channel methods of communication could be implemented into code so that information can be passed out from an organization undetected. The scenarios are boundless.

Malware can incorporate various techniques to not only avoid forensic detection, but can also avoid forensic analysis. Grugq (n.d.) refers to this as “anti-forensics” and suggests that there are 3 fundamental ways of achieving this. Namely:

- Data Destruction
- Data Hiding
- Data Contraception

Essentially data destruction means deleting file residue such as inodes, directory entries and data blocks. It also includes deleting file system activity though inode time stamps. Data hiding means putting data where it is not expected to be. This can include storing data in blocks that have been marked as bad. Data contraception means making sure data is never stored on the disk. Grugq implies that it is better to never have created the data in the first place than having to destroy it.

Forensic analysis avoidance techniques can include, but are not limited to:

- Exploiting flaws in analysis tools.
- Attempting system destruction when being analysed.
- Generation of false disassembly listings.
- Subversion of disassembler heuristics.

This paper analyses these techniques to show how they can be recognised so that forensic analysis can be performed. This treatment is far from exhaustive, and serves merely as an introduction to the techniques employed to avoid forensic analysis of malicious software. The background section of this paper shows how reverse engineering techniques are typically used for the forensic analysis of malware binaries. This analysis can be either static or active. Static analysis is used to analyse the binary without executing it, whilst active analysis requires the execution of the binary and studying its behaviour. This paper then moves into how such reverse engineering analysis techniques can be avoided by the malware writer. The final section of this paper then looks at automated malware classification techniques that are currently being developed. This is because commercial virus scanners will be very unlikely to detect the presence of customized, or purpose built malware.

For the purpose of this paper, legal issues surrounding the reverse engineering of software is ignored. It is assumed that the software that is being examined is the intellectual property of the organization that will have authorised this activity, or that the investigative team is authorised to conduct the investigation in some other way.

BACKGROUND

Evidence Elimination

Penalities for hacking and writing malware are increasing. Awareness of hacking is rising within the community as well as forensic analysis techniques and methodologies. It is in the interest of hackers and malware writers to work against forensic investigators and eliminate forensic evidence. They have the motivation, the opportunity and now the methods to defeat forensic analysis. Kotadia (2006) reported in an online article that a speaker at the IT Security in Government Conference held in Canberra in July 2006 claimed that 65% of new malware "uses some type of stealth or anti-forensic technology in an attempt to remain undetected before, during and after an attack".

Static Analysis

Various techniques used for static analysis include "information gathering, disassembly, symbol table regeneration, decompilation techniques, and methodologies for determining the order of decompiling subroutines" (The Honeynet Project, 2004, p.452). These techniques are discussed in the following paragraphs.

Information Gathering

Information can be gathered from a binary using various tools. The "file" command can provide the file format and identify the target platform. Use of the "ldd" command can give information on dynamic link libraries required for the code to run. Embedded strings can be found in the binaries by using the "strings" command. The "hexdump" command with appropriate switches can do this too, as well as listing the name and version of the compiler used. Symbols can be listed by using the "nm" command. The language used to write the program can be determined by various characteristics found in the binary. This can include the way the strings in the binary are terminated, how data arrays are stored, how subroutines are called and the order of storing parameters on the stack.

Disassembly

The process of disassembly is used to create a more readable version of the binary. Typical programs in the Linux environment include ndisasm and objdump. Peikari and Chuvakin (2004, p.51) point out that objdump provides a sequential disassembly, and "no attempt is made to reconstruct the flow of the target". It also cannot cope with binaries that have had their section headers removed, or that are invalid. This can be done by the tool "sstrip" which is discussed later. The Honeynet project (2004, p. 456) also point out that compiled programs usually put the data in the data segment, and the code in the code segment, but it is possible that data can be treated like code and code may be treated as data. Heuristics are used by disassemblers to separate code from data, and more professional disassemblers such as IDA Pro can trace program flow by using signature based heuristics.

Symbol Table Regeneration

The “nm” command can be used in the linux environment to list symbols in object files. This is useful because the addresses of functions in programs can then be located and then analysed. More than likely, the malicious software writer will have used the “strip” command to remove all symbols from the object file. Programs can be either statically or dynamically linked. If dynamically linked, external library calls can be identified, however, if statically linked, all external routines are combined into the binary when compiled. This makes it impossible to distinguish external routines from linked libraries from the routines written by the programmer. The Honeynet Project (2004, p.458) say that “An ideal solution would be to recreate a symbol table containing the mapping of library names to addresses so that only the user code needs to be analysed”. They go on to say that a database of signatures could be created from subroutines from all known libraries. Then, to recreate the symbol table and insert it back into the program, the signatures could be matched against the code being examined. Then, external routines can be factored out, leaving only the code written by the malware writer to be examined.

Decompilation Techniques

Decompilation is the action of transforming the results of a disassembly back into a high level language. The process works basically by understanding how particular compilers work in the first place to generate the object code, with particular consideration of how optimisations can take place. After disassembly, the entry points to all functions can be determined. Usually this is done by searching the disassembly for calls to absolute addresses. Subroutines can also be identified by common header and footer signatures. Analysis of each subroutine can then begin. This can consist of recognising flow control structures such as loops, conditional evaluation and branches. Parameters passed to and from these routines can also be determined, as well as other statements and assignments.

Active Analysis

Active analysis techniques listed by the Honeynet Project (2004, p. 464) include sandboxing the analysis environment, blackbox analysis and tracing. These techniques are discussed below.

Sandboxing the Analysis Environment

The Honeynet Project (2004, p.464) emphasises that “It would be foolish to execute the program on a production system or one connected to a live network”. They go on to recommend that the ideal place to perform active analysis would be on the honeynet on which it was captured from. It would be expected that local and remote logging would be in place already, and network packet capture is available. The analyst should have super user privileges so that kernel modules can be loaded and unloaded, system logs can be examined, file system images saved and restored and routing tables modified. They also recommend a wide variety of debuggers are available. This is because if different debuggers give different results, it would indicate that the malware is most likely targeting specific debuggers with avoidance techniques.

Other types of sandboxes include hard and soft virtual machines. A typical soft virtual machine is vmware which allows the investigator to take snapshots of the system state at any time and restore that particular state. This makes it very easy to repeat an analysis from a known point. Another advantage is that vmware allows the establishment of virtual networks, that can be confined from the host. This enables the investigator to observe the network behaviour of malware in a sandboxed environment.

Blackbox Analysis

Blackbox testing in software engineering means testing software without knowing how the internals work. The focus is on state, by varying inputs and recording outputs. If a program is in an endless loop, it could indicate that the process is a daemon. Possibly the process is waiting for a network event such as a connection. It should also be possible to determine which files have been accessed by the process by examining file system states and access time stamps. It should also be possible to capture network packets using tools such as ethereal. It should also be possible to determine if any child processes were forked.

Tracing

Tracing provides a more internal view than does blackbox analysis by tracing system calls, library calls and internal calls. Linux provides the “strace” program that is used. “strace works by intercepting and recording the system calls used by a process and the signals it receives. For each system call, the name, arguments and return value are printed to standard error (stderr) or to a file specified” Frye (2005). This is really useful, because it is possible to see the names and paths of files accessed and in which mode, system calls, shared libraries it needs, as well as many other parameters. It is possible too to attach to a running process

DETECTION AND ANALYSIS AVOIDANCE

Obfuscation

A paper by Christoderescu and Jha (2003) show how effectively obfuscation techniques such as dead-code insertion, code transposition and instruction substitution can defeat commercial virus scanners. One of the simple tests was to insert nop codes into the Chernobyl virus. They show that the Norton Anti-Virus software did not detect this obfuscated virus.

Two common techniques used by malware writers to avoid detection through obfuscation are polymorphism and metamorphism (Christoderescu, Jha, Seisha, Song, Bryant, n.d., p.1). A virus can use polymorphism to avoid detection by decrypting its encrypted malicious payload when it is being executed. “A polymorphic virus obfuscates its decryption loop using several transformations, such as nop-insertion, code transposition (changing the order of instructions and placing jump instructions to maintain the original semantics), and register reassignment (permuting the register allocation)” (Christoderescu et.al., n.d., p.1.). A metamorphic virus changes the code a number of different ways when it replicates. This can include code transposition, changing conditional jumps, register reassignment and substitution of equivalent instruction sequences.

These techniques need not remain in the domain of self replicating viruses. These techniques can be used in any malicious software to hide its presence through obfuscation. Most worms and viruses are variations of an original virus or worm and signatures are soon determined once they have been detected in the wild. There is no doubt that these techniques can be used by the hacker to create a single version instance of malware customised to perform some malicious act within the company in which they are employed as a software or network engineer. They may use these techniques to deliberately obfuscate its function. The 2003 paper by Christoderescu et.al. points out that most malware detectors are easily defeated by obfuscation because they use pattern matching and that they are not resilient to slight variations. They go on to say that pattern matching ignores the semantics of instructions.

By being able to run a semantics focused parser over suspicious binaries under forensic examination, it could be possible to determine which binaries have a malicious intent. It could help the forensics investigator to filter out binaries of interest faster. The sections that follow discuss techniques used to avoid detection and analysis. Understanding the basic mechanics of these techniques will not only help the forensic analyst detect these techniques but should also assist in developing automated malware classification programs.

Anti-disassembly

By knowing that disassemblers use signature based heuristics, malware can be written to trick a particular disassembler into producing an incorrect disassembly for the binary.

Library calls can be made to be hard to identify by using static linking. The disassembler has to be able to match library function signatures.

A paper written by Linn and Debray (n.d.) describes algorithms that can be used to improve resistance of programs to static disassembly through obfuscation. Their focus is on providing security to protect intellectual property, stop piracy, and identification of breaches through making the disassembling process difficult. The same principles can be used by the malware writer equally well to hide their own intentions. Linn et.al.’s goal is to make the cost of reconstructing the high level structure too high, which is exactly the motivation of the malware writer as well. They say “In order to thwart a disassembler, we have to somehow confuse, as much as possible, its notion of where the instruction boundaries in a program lie” (Linn et.al., n.d., p.3). Methods discussed in their paper include introducing disassembly errors that “repair themselves” and injection of “junk bytes” into the instruction stream. These techniques are reliant upon the type of instruction set of the processor, and on the type of sweep of the disassembler. That is whether the disassembler performs a linear sweep or a recursive traversal. A weakness of a linear sweep algorithm is that it is subject to errors if data is embedded in the instruction stream. Such a disassembler is aware of disassembly errors only if an invalid opcode is found. It

does not allow for the control flow behaviour of the code. A recursive traversal algorithm does take into account the control flow behaviour of the code. However a "weakness is that its key assumption, that we can precisely identify the set of control flow successors of each control transfer operation in the program, may not always hold in the case of indirect jumps" (Linn et.al. n.d., p.3). Another technique they use for inserting junk bytes is what they refer to as jump table spoofing. They do this to mislead recursive traversal disassembly. Fundamentally, the idea is to make the code addresses in the jump table to addresses in the text segment, which do not correspond to instructions and hence create disassembly errors.

Encrypted Data

Packers

Miras and Steele (2005, p.20) explain that packers compress executable programs in the objective of making the task of reverse engineering as difficult as possible. A packed program contains a stub that is used for decompressing the compressed data. The executable is decompressed during loading. Popular packers used in the Windows environment that use PE format files include PECompact, UPX, ASPack and Armadillo. Packers use various techniques to subvert forensic analysis. These include:

- Built in anti-debugging.
- Insertion of junk code.
- Abuse of exception coding.
- Trick disassemblers by jumping into longer instructions.

Packer detection methods include examining signature and heuristics. An effective signature based method is to compare the program entry point against a database. Heuristical methods include seeing how the byte distribution (entropy) is changed by the packers as well as checking import tables.

Burneye

Burneye "is an executable and linking format (ELF) encryption tool that limits forensics tools' reverse-engineering capabilities by protecting the binary program. Burneye users can manipulate executable code so only an attacker with a password can run the program" (Saita, 2003). A loadable kernel module called burndump can be used to remove burneye from binaries. Another tool that detects the use of burneye is called bindview. Burncrack is a cracker that works with the John the Ripper program that can crack and unwrap burneye protected binaries without having to run them. This would be a very useful tool when required to perform a forensic analysis of the malware.

Data Destruction

The purpose behind data destruction is to leave nothing useful for a forensics investigator, effectively removing all trace of evidence. The Defiler's Toolkit is a set of programs whose purpose is to prevent forensic analysis, specifically targeting the ext2fs filesystem, commonly found on linux systems. Necrofile is one of the programs on the Defiler's Toolkit for this purpose. Ordinarily, when a file is deleted, the inode and directory entries, known as the metadata are left untouched. A forensic investigator will look at the metadata to see if the supposedly erased data can be recovered. Necrofile can remove this metadata making it extremely difficult for the investigator to recover files. Klismafile is another program in the toolkit that removes directory entries of filenames that have been deleted. Through the use of these programs, forensic evidence can be removed. It would not be inconceivable for malicious code to perform these two actions autonomously if it detected that forensic analysis was being perfomed.

Data Hiding

The purpose of data hiding is to hide evidence from the forensic investigator, and is only successful if the investigator does not know where to look for the evidence.

In the past, knowing that tools such as The Coroner's Toolkit (TCT) did not look at bad blocks on a disk drive that was using the Second Extended File System (ext2fs), an attacker could use the bad blocks inode to include good blocks, and hide data there. Ordinarily, the bad blocks inode only points to bad blocks, and these blocks will not be used for files. It is advisable to make sure that TCT's more recent version (TASK) is used and that bad blocks on a disk are also investigated (Skoudis, 2003). There is no doubt that this is a bit dated, but the point should be clear that flaws can be found in the forensics tools, and most likely will continue to be found as tools are improved and developed.

Data Contraception

Grugq (n.d.) says that the two core principles of data contraception are to prevent data from being written to disk, operating purely in memory and to use common tools rather than custom tools. The idea is to limit the value of any evidence that does touch the disk. Rootkits can operate in memory and "use ptrace() to attach to an existing process and inject code into it's address space. Additionally, injecting kernel modules directly into the kernel is also a well known technique" (Grugq, n.d.).

Grugq recommends using common utilities such as rexec, which remotely executes a command on a remote host. This allows the malware or hacker to never have to write anything to disk.

Antidebugging

A common debugger used in the linux environment is gdb. If a program has been compiled with the option to produce debugging information, it could be possible to hide malicious intent, such as a logic bomb. The following code, ptrace.c demonstrates this.

```
#include <sys/ptrace.h>
#include <stdio.h>
int main()
{
        if  (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)  {
                printf("Debugging detected, goodbye!\n");
                return 1;
        }
        printf('Malicious purpose here.");
        return 0;
}
```

The program is compiled with the following command:

gcc –o ptrace ptrace.c –g

When run at the command line, the program will print the following line.

Malicious purpose here.

When run inside gdb, it will print

Debugging detected, goodbye!

Hopefully, this sort of thing will be picked up in a peer review, or some other audit activity, but could be missed by an inexperienced auditor, or an auditor not focussing on security. The code could even be in a library that is not in version control and is going to be linked in, in a nightly build. It is possible to hide the approach even more by replacing the call to ptrace with an int 80 system call with inline assembly (Peikari et.al., 2004, p.69).

The ptrace trick can be overcome by setting breakpoints at the start and the end of the subroutine that calls ptrace (The Honeynet Project, 2004, p.469). Then when running in the debugger, the return value from the call to ptrace is changed from failure to success.

Compression Bombs

Compression bombs "are files that have been repeatedly compressed, typically dozens or hundreds of times" (NIST, 2006, p.45). They can cause forensic tools used for examination to fail, or use up so many resources that the computer may hang or crash. It is possible that they may contain a malicious payload. NIST (2006, p45) suggest it could be very difficult to detect a compression bomb before it is uncompressed. However it is very important to be working on the forensic image, so that the system can be restored if required. It is also suggested by NIST (2006, p45) that a virus scanner may detect the bomb. But this is reliant upon signatures being kept up to date, and also that the bomb has been in the wild for a while, and that anti virus companies have analysed it before. More of a concern, would be a one off, customised compression bomb that would not be recognised by its signature and could be effected once malicious software determines that an investigation is taking place.

AUTOMATED MALWARE DETECTION AND CLASSIFICATION

Malware detection and analysis by an investigator can be a labor intensive process using static and active techniques. Due to time constraints and the abilities of the investigator, there is a possibility that critical forensic evidence could be overlooked. To this end, automated malware detection and classification tools are being developed.

A presentation by Bilar (2005) shows how malware can be classified by analyzing statistical structures. Three perspectives are examined by Bilar, including Assembly Instructions, Win 32 API Calls and System Dependence Graphs. Examination of Assembly Instructions is primarily a static analysis technique where the frequency distribution of opcodes are developed from the disassembly of the binary. Bilar shows that this technique can be useful to provide a quick identification. Just looking at the most frequent opcodes is a weak predictor. Looking at fourteen of the most infrequently used opcodes such as int and nop it may be possible to classify malware. Bilar suggests that root kits make heavy use of software interrupts, and viruses make use of nop for padding sleds. Additional work being carried out in this area includes investigating equivalent opcode substitution effects, and association effects between compilers and types of opcodes.

Tracking Win 32 API Calls is an active analysis technique that observes the API calls that a program under investigation makes. These calls are recorded and a count vector is saved into a database. These vectors are then compared to known malware vectors in the database, and it is determined if the vectors are related. Bilar (p.25) claims that this vector classification is successful in classification of malware into a family. The Win 32 API call fingerprint is shown by Bilar (p.27) to be robust, even though various packers were used.

System Dependence Graphs is a newly developing static analysis technique described by Bilar (p.31) that represents control, call and data dependencies of a program through graph modeling. Then graph structures can be used as fingerprints, which assist in the process of identification, classification and prediction of behaviour.

Lee and Mody "propose an automated classification method based on runtime behavioral data and machine learning" (2006, p.3). Essentially the run time behaviour of a file is represented by a sequence of events which is stored in a canonical format in a database. Machine learning is used to recognize patterns and similarities which are then used to classify new objects.

CONCLUSION:

This paper has shown how forensic analysis can be avoided using techniques used by the writers of malicious software. It has been reported that malware is increasingly using these techniques to avoid forensic analysis, and that the majority of new malware is incorporating these techniques. A number of avoidance techniques were discussed including obfuscation, anti-disassembly, encrypted and compressed data, data destruction and anti-debugging. This list of techniques is far from exhaustive. These techniques need not remain in the domain of mass distributed malware such as worms and viruses. They could be developed primarily for a one off, malicious mission of performing some malicious act within a company, collection of commercially sensitive data or other equally criminal activities.

Automated detection and classification work is progressing in this field. This includes analysing statistical structures such as assembly instructions, system calls, system dependence graphs and classification through machine learning.

REFERENCES:

Bilar, D., (2005). *Statistical Structures: Fingerprinting Malware for Classification and Analysis*. Retrieved September 2, 2006 From www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bilar.pdf

Christoderescu, M., Jha, S., Seisha, S.A., Song, D., Bryant, R.E., (date unknown). *Semantics-Aware Malware Detection*. Retrieved August 30, 2006 From http://www.cs.cmu.edu/~bryant/pubdir/oakland05.pdf#search=%22malware%20%2Bobfuscation%22

Christoderescu, M., Jha, S., (2003). *Static Analysis of Executables to Detect Malicious Patterns*. Retrieved September 2, 2006 From www.cs.cornell.edu/courses/cs711/2005fa/papers/cj-usenix03.pdf

Frye, M., (August 2005). *Debugging codes with strace*. Retrieved August 26 2006 from www.redhat.com/magazine/010aug05/features/strace/

Grugq, (date unknown). *The Art of Defiling, Defeating Forensic Analysis on Unix File Systems.* Retrieved August 26, 2006 from http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-asia-03/bh-asia-03-grugq/bh-asia-03-grugq.pdf

Kotadia, M., (28 July 2006). *Beware 'suicidal' malware, says CyberTrust.* Retrieved August 27, 2006 from http://software.silicon.com/malware/0,3800003100,39160966,00.htm

Linn, C., Debray, S., (date unknown). *Obfuscation of Executable Code to Improve Resistance to Static Disassembly*. Retrieved August 30, 2006 From www.cs.arizona.edu/solar/papers/CCS2003.pdf

Miras, L., Steele, K., (September 2005). *Static Malware Detection*. Retrieved September 2, 2006 From http://www.toorcon.org/2005/slides/lmiraksteele-staticmalwaredetection.pdf#search=%22malware%20%2Bpacker%22

NIST, (August 2006). *Guide to Integrating Forensic Techniques into Incident Response*. Retrieved September 1, 2006 From http://csrc.nist.gov/publications/nistpubs/800-86/SP800-86.pdf

Phrack Inc, (date unknown). *Defeating Forensic Analysis on Unix.* Retrieved August 27 2006 from http://www.theparticle.com/files/txt/hacking/phrack/p59-0x06.txt

Saita, A., (May 2003). *Antiforensics: The Looming Arms Race.* Retrieved August 26, 2006 from http://infosecuritymag.techtarget.com/2003/may/antiforensics.shtml

Skoudis, E., (June 6 2003). *Breaking News – The Latest Computer Attacks and Defences*. Retrieved August 27 2006 from www.counterhack.net/UFL.ppt

The Honeynet Project, (2004). *Know Your Enemy Learning About Security Threats.* Addison-Wesley, Boston

COPYRIGHT

[Murray Brand] ©2007. The author/s assign SCISSEC & Edith Cowan University a non-exclusive license to use this document for personal use provided that the article is used in full and this copyright statement is reproduced. The authors also grant a non-exclusive license to SCISSEC & ECU to publish this document in full in the Conference Proceedings. Such documents may be published on the World Wide Web, CD-ROM, in printed form, and on mirror sites on the World Wide Web. Any other usage is prohibited without the express permission of the authors