An Instrumented Analysis of Unknown Software and Malware Driven by Free Libre Open Source Software

Gérard Wagener
University of Luxembourg
SES ASTRA
gerard.wagener@ses-astra.com

Alexandre Dulaunoy
SES ASTRA
a@foo.be

Thomas Engel
University of Luxembourg
Thomas.Engel@uni.lu

Abstract

Reverse engineering is often the last resort for analyzing unknown or closed source software. Such an investigation is motivated by a risk evaluation of closed source programs or by evaluating consequences and countermeasures against infections by malicious programs that are often closed source. This article presents a success story where we used and modified free software serving as environment for analyzing unknown software. We explain how a malware sandbox can be constructed based on free software. Moreover we describe how we modified free software to improve malware analysis with additional features or extensions. Free software helped us to increase the accuracy of malware or unknown software analysis.

1 Introduction

Key words like confidentiality, access-control, integrity, authenticity resurface regularly in a security context [24]. For free software, four approaches can be used to verify if these key words are justified. Firstly, the source code can be read and tested. A second possibility is to follow the open discussions in public mailing lists. A third possibility is to read change logs and a forth possibility is to read the logs of the public version system (i.e. *cvs* [5], *subversion* [8], *git* [25]) hosting the piece of software. After having understood the source code, a formal verification can be done to ensure assumptions. A third-party can also conduct such an audit.

For closed source solutions such a verification is more difficult. The above mentioned verification approaches are privileges of the person having the authorized access. A non-disclosure agreement can be done with third-parties [3]. Often these agreements are non-transitive. Each subcontractor must sign a non-disclosure agreement with the person having the author rights. Of course there remains the possibility of reverse engineering based on security tools, like disassemblers, debuggers, virtual machines and alike. Even if a closed source software customer is aware of vulnerabilities he or she must wait until the software vendor provides a new software release. In the free software world, the customer can patch the program and publicly divulge the patch.

An increase of malicious software is predicted [20]. Malicious software is often denoted as malware. Malware is software with malicious intent and a wide spectrum of malicious programs can be found in the wild [2]. Malicious programs usually penetrate machines without the user's agreement. Another common infection technique is that a user installs a trojan horse. Families of malicious programs spy the user's activities and send private and confidential data to third-parties without the user's awareness. Other families hijack users' resources for attacking other machines. Antivirus programs are used for detecting malicious software and for preventing the damage. These capabilities are implemented by the antivirus community which collects malicious closed source programs and reverse engineers them with the purpose of protecting other users.

This article focuses on the analysis of malicious binary programs that are closed source and run in a Microsoft Windows operating system. For such programs, reverse engineering tools are needed to find countermeasures, as explained in section 2, followed by limits of closed source analysis presented in section 3. Our successful experiences with malicious software analysis driven by free software are described in section 4. We profited from free software freedoms aiming to improve analysis of malicious programs, summarized in section 5 and the limits are announced in section 6. The related work is outlined in section 7 followed by future work activities motivated in section 8.

1.1 Terms and definitions

In this article we reuse the **F**ree-**L**ibre / **O**pen **S**ource **S**oftware definition relying on the Free Software Movement [12, 33, 9].

1.1.1 FLOSS

FLOSS stands for **F**ree-**L**ibre / **O**pen **S**ource **S**oftware. It characterizes software licenses which provide all the four freedoms $\mathcal{F} = \{F_0, F_1, F_2, F_3\}$, like it is described in the GNU project [12]. Only a software tool, or a software library, that provides all the four freedoms in $\mathcal{F}$ is considered as "free" in this article.

F_0 A program can be run for any purpose.

F_1 The source code can be studied in order to determine how a program works.

F_2 Copies of the program can be distributed.

F_3 Anybody can improve the program and publish releases.

A consequence of the F_0 freedom is that a program can be run for research or operational purposes. Studying the internals of a program, possible through the F_1 freedom, helps to understand a program and to evaluate the risk of using the program. The F_2 freedom improves the dissemination of a program. When a program is not supported anymore, someone else can take over the program, improve it and share the improvements, based on the freedom F_3.

1.1.2 Closed source software

A frequently used synonym for closed software is proprietary software. Generally, at least one freedom $f \in \mathcal{F}$ is not granted. In this article, if not specified, we consider closed source software as software where at least the freedom F_1 is prohibited.

2 Malware analysis

2.1 Discussion

According to the author of [1], a personal computer is a fundamentally insecure platform due to its open and accessible architecture. A threat originates from software running on the platform. Thus the author proposes code protection techniques aiming to protect software, founded on cryptography and anti-reverse engineering techniques. Another reason for protecting software is to avoid software piracy and intellectual property violations [17].

Software protection is a two-edged sword because authors of malicious software also apply such techniques [10, 29, 31]. They try to keep their source code secret aiming to protect them against antivirus software or other malware authors. Once the internals of a malware sample are revealed, the end of its life cycle is reached. A continuous race is established between malware authors and malware analysts.

2.2 Delegation of malware analysis

Strong reverse engineering skills and considerable efforts is required for malware analysis. Automated malware analysis services are offered by companies. A customer can submit a malware sample to such a service and later receives a report of its internals. The sample is often put in a sandbox at the company side, where observations are done. The Norman sandbox was developed by the company Norman [18] and the CWSandbox [34], developed by the university of Mannheim, was acquired by the company Sunbelt.

2.3 Analysis tools off the shelf

Due to the fact that the source code is not available for closed source software and reverse engineering is the last resort, disassemblers, debuggers, virtual machines and monitoring tools belong to the toolbox of the reverse engineer. Two malware analysis approaches are usually applied. The first approach is called static analysis. In this case the malicious program is not executed, only its machine code is transformed in assembler code with disassemblers and studied. The second approach is called dynamic analysis, where a malicious program is executed and observed [29, 34]. Debuggers, virtual machines and monitoring tools are used for this purpose. A pleasant phenomenon is that most free reverse engineering tools / libraries are included in a GNU/Linux distribution and the below mentioned tools / libraries are included in Ubuntu 8.04. Frequently reverse engineering tools follow a standard unix design [21] and can be used by a command line interface. Thus they can be used by scripts or front-ends.

2.3.1 Disassemblers

Disassemblers are very popular for mapping machine code to assembler code, that is studied later on. A notable disassembler is a proprietary disassembler called *IDA Pro* [13]. *IDA Pro* has some powerful features like recognizing system functions, extracting and presenting visually control flow graphs and debugging a subset of machine instructions.

The free tool *objdump* can also be used to disassemble programs, with a graphical front-end called *dissy*. The tool *dissy* provides a graphical representation of the disassembled jump instructions. Unfortunately this disassembler is vulnerable to linear sweep attack [17]. Free disassembler libraries like *libdisasm* can be integrated in other free software. Free disassemblers are often combined with hex editors, as it is with case in the free tool *ht*.

2.3.2 Virtual machines

Malicious programs can be executed in virtual machines and observed aiming to mitigate disassembler attacks [19].

Usually high-level information is gathered from malware samples, like file system changes, registry changes and network activities. Using virtual machines it is easy to record the state just before the malicious program execution and the state of the virtual machine after the execution using snapshots [19]. These snapshots can then be compared. In practice the malware sample is executed for a few seconds to a few minutes and then killed. An alternative to the closed source virtual machine *VMware* is *qemu* [6].

Besides virtual machines, virtual operating systems were developed. In the User-Mode Linux project, the Linux kernel is instrumented to run in user space. Thus multiple Linux operating systems can run on top of a Linux system.

2.3.3 Debuggers

Sometimes researchers are interested in intermediate states of a malware execution. Debuggers can be used for this purpose, *softice* is a noteworthy proprietary kernel debugger [19]. Without any anti-debugging techniques, the execution of a program can be stopped, at any time, the memory and the processor state can be inspected. A popular free debugger is *gdb* which is followed by many graphical front-ends like *ddd* and *cdbg*.

2.3.4 Software monitors

Besides debuggers and virtual machines, other monitoring techniques were explored. In a Microsoft Windows operating system, functions of external libraries can be redirected with a proprietary library called *detours* [14]. Specialized kernel functions can also be diverted and monitored. One software component runs in kernel space and monitors tasks. The other one runs in user space and reports observations. Proprietary examples of such monitoring tools are *filemon* [22] and *regmon* [23]. Free alternatives running in a Linux operating system are *strace*, *ptrace* and *ltrace*.

3 Limits of malware analysis

3.1 Fundamental limits

Static and dynamic analysis, presented in section 2.3, have fundamental limits. During static analysis, the control flow may depend on dynamic variables, hardly possible to determine in advance without emulation. Moreover instructions can be generated during execution and then executed which provoke a control-flow graph change during execution [29]. For the dynamic analysis it was proved, based on Alan Turing's Halting Problem, that it is undecidable to foresee whether a malicious program has finished [10].

3.2 Limits of closed approaches

Malware authors often create programs capable of evading the analysis process. For this purpose they often exploit ambiguities or artifacts of the analysis tools. Virtual machines often have unique hardware strings which are queried by malware samples for the purpose of detection [19]. Binary patches for the closed source virtual machine *VMware* were created, targeting a better camouflage [16].

For free software solutions such changes are easier to perform, due to the freedoms F_1 and F_3. Other detection techniques exploit incomplete implementations. Often specialized processor instructions are not implemented in virtual machines, executed by malicious programs which often ends up in the end of the analysis. Free software can be studied in order to detect such vulnerabilities and patches can be written.

Malicious programs sometimes pretend to be the monitoring program in user space and thus are able to detect the kernel space component if a successful communication is observed. This becomes possible due to the fact that most closed source monitoring program versions are using a constant device name. Users have the ability to change such constant artifacts in free tools and thus make such a detection technique more difficult.

Due to the fact that on a standard PC programs are running concurrently, the monitoring tool or the user must be able to decide which observations belong to concurrent programs and which ones belong to the targeted monitored program. Malicious programs often create multiple processes or generate machine instructions in memory regions where normally no instructions should be and execute the instructions from there [29]. With this behavior malicious programs hope to evade the monitoring process. In order to evaluate the accuracy of the monitoring program it is mandatory to check (i) if an observation filtering is done and (ii) how it is done. For free software these checks can be easily done, enabled by the freedom F_1. Checksums on the machine instructions level are a very powerful technique for detecting interference with monitoring tools [29]. A solution to this problem is to not modify the memory of the malicious program, but its environment which is possible with free software.

Techniques were elaborated to detect proprietary sandboxes [26]. The issue with a malware analysis delegation approach is still a closed-approach for software analysis. Beside the openness of the API[1] to access it, the inner working of such malware analysis is still unknown or described in a high-level way. It is important to have access to the source code of such sandboxes in order to evaluate the accuracy of the analysis.

[1] Application Programming Interface

Table 1. Disassembler attacks

	$IDAPro^{\Diamond}$	$objdump^{\heartsuit}$	$ndisasm^{\heartsuit}$	$dissy^{\heartsuit}$
$strip$	$\times$	$\times$	$\times$	$\surd$
ls_0	$\times$	$\surd$	$\surd$	$\surd$
dt	$\surd$	$\emptyset$	$\emptyset$	$\surd$

$\Diamond$ Closed source $\heartsuit$ Free software $\emptyset$ Not applicable
$\surd$ Attack success $\times$ Attack failure

Table 2. Debugger attacks

	$gdb^{\heartsuit}$	$softice^{\Diamond}$
int3	$\surd$	$\surd$
device	$\times$	$\surd$

Symbols are reused from table 1

4 Malware analysis driven by FLOSS

In our previous research activities we tested some security analysis tools and evaluated various anti-reverse engineering techniques [29]. We gradually modified free software aiming to improve malware analysis. Firstly, we developed a free malware sandbox, based on existing free software which is able to extract high-level information from unknown malicious software [29], in a similar way to conventional proprietary sandboxes like the Norman sandbox and CWSandbox. We improved our sandbox to extract system function calls targeting a malware classification [31]. Finally, we transformed our sandbox into a high-level debugger for the purpose of malware reverse engineering.

4.1 Experiments

We wrote some anti-reverse engineering traps and analyzed our code against closed source and free reverse engineering tools. Table 1 shows results of disassembler attacks described in [29] and table 2 presents results of debugger detection techniques, explained in [29].

Table 1 and 2 show that closed source and free tools for analysis of closed source have their limitations and that free software does not provide a warranty of always being better than closed source software. Nevertheless free software provides the flexibility of fixing discovered limitations and does not require a response from the original author. If the quality is not sufficient one can always improve the tools which is not possible for closed source software.

5 Improving results by FLOSS modification

We now describe how we used and changed other free tools for malware analysis. Our goal was to create a fake environment, based on free tools, in which to execute and observe closed source malicious software. This environment is a framework of free software called **A**utomated **A**nalysis and **N**etwork **E**mulation (ANNE) [29, 27].

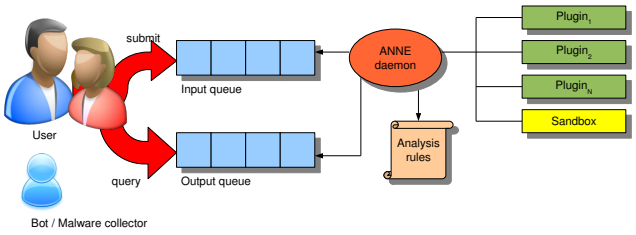

Figure 1. ANNE framework architecture

5.1 Requirements

We target reverse engineering based on freedom F_0. Next the freedom F_1 allows us to evaluate the risk of our framework in depth. This is a mandatory requirement to best control the framework. Moreover the freedom F_2 permits us to share our framework, composed of free software among other malware researchers. We do not want to offer a service to third-parties but we want to make the framework freely available. Thus other researchers can judge and extend the framework themselves and have not to rely on service descriptions. Malware authors often quickly find anti-reverse-engineering techniques. Therefore it is mandatory to be not dependent on third-parties to create countermeasures by code modifications. The freedom F_3 permits this solution.

5.2 ANNE architecture

A generic architecture is presented in figure 1. Users or robots can submit malware samples in a persistent input queue. In doing so the user gets a unique identifier of their job. This unique identifier can be used to query the analysis. The daemon polls the queue and performs malware analysis and puts results in the output queue. The user can get results by providing their unique job id. Proprietary sandbox services ship analysis results via email. Moreover they are protected with security codes in order to make robots and queuing attacks inefficient. A free sandbox can be set up in environments where only trusted users can use it, in a distributed fashion. The sandbox can be operated not only at one company but at several institutions. Thus queuing attacks and denial of service attacks can be mitigated.

The daemon takes a malware sample and processes it according to user-defined rules. The analysis task is divided

into different subtasks $\{1, \ldots, N\}$. Each subtask is done by a plug-in. Each plug-in follows the standard unix philosophy. It takes command-line parameters and communicates via standard input, standard output and standard error. An example of a plug-in is the static disassembler *objdump* which is encapsulated in a wrapper script that adopts the output format. An other example is a plug-inthat computes the entropy of the binary giving an indication of encrypted or packed binaries. Until now the most powerful plug-in to the ANNE architecture was the sandbox.

5.3 Usage of side effects of free software tools

In this section we explain how the freedom F_0 is a benefit as no fixed usage of the software is prescribed. Initially the tool *wine* was developed aiming to execute Windows binaries on top of a Linux operating system. The rationale why we chose *wine* to execute malware samples is that we have full access to the source code which is not the case for a native operating Microsoft operating system. Full access to the source provides a greater flexibility than using a public API.

After having inspected the source code of the components, we noticed that *wine*'s file system and process management and networking capabilities should be mitigated. We did the mitigation by installing *wine* in a User-Mode Linux which is connected via a virtual network with the host operating system. The developers of *wine* created an elaborated debugging system. From this debug system we gather information about file system and registry access. The sandbox program is a standard Linux process that is executed via the daemon. The sandbox takes a malicious program as command-line argument along with a timeout value. Due to the Halting Problem, we do not know when the malware sample finishes. Therefore we kill it after the timeout expired. After having examined analysis results, the user can trigger a new analysis with a different timeout value. In practice we execute a malicious program for one minute and if we do not observe sufficient actions we then execute it for five minutes.

The sandbox process copies the malware sample in the User-Mode Linux, tagged in figure 2 as **(1)**. The local controller in the User-Mode Linux is then executed via SSH **(2)**. The local controller ensures that *wine*'s environment is clean, by restoring the initial file system and registry and moves the malware sample inside **(3)**. The local controller enables the debug mode of the tool *wine* and then starts it **(4)**. Inside the User-Mode Linux was a DNS[2] server and a TCP[3] server. The default DNS server points to the local DNS server. In case the malware sample makes a DNS

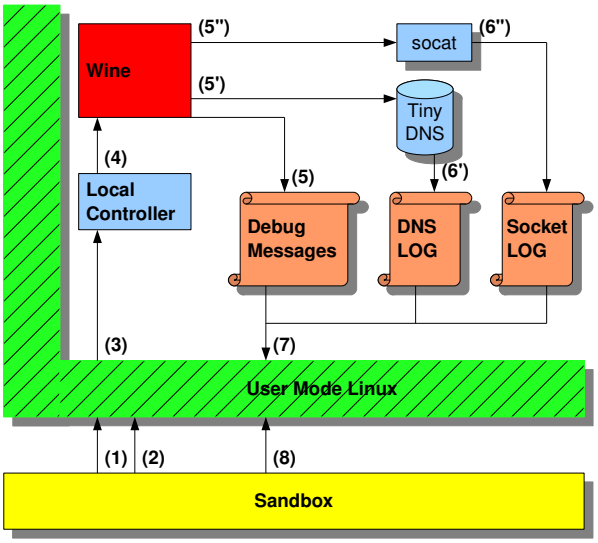

Figure 2. Sandbox

query **(5')** it connects to our local fake TCP server **(5")** using the resolved IP address. Thus we can capture the first bytes of the malware sample's communications. After a timeout the local controller kills *wine* and compares *wine*'s file system with the initial one aiming to see file system and registry changes. The debug messages of *wine*, the DNS logs and the captured TCP communications are stored **(7)**. The sandbox plug-in recovers *wine*'s debug messages, the DNS logs and the TCP communications via ssh[4] **(8)**. Again here we see that the freedom F_0 allows non standard usage of free software. With such a free software instrumentation we are able to extract the following information from a malware sample:

- File system changes (by comparing file systems)
- File access (from debug messages)
- Registry changes (by comparing files)
- Registry access (from debug messages)
- DNS queries (by our DNS server)
- Initial bytes from a socket (by a fake TCP server)

5.4 Changing the scope of free software

After having described a proof of concept of a malware sandbox, in section 5.2 which is based wholly on free software, we propose mechanisms on how the sandbox can be

[2]Domain Name System
[3]Transmission Control Protocol
[4]Secure Shell

improved by modifying the source code of the components profiting from the freedom F_3.

5.4.1 Improving the interception of malware communication

The sandbox presented in section 5.2 suffers from various drawbacks. In case the malicious program does not use the DNS service, the sent network packets do not reach our TCP server. Moreover we used standard ports on which the TCP server listened. When a malicious program uses non-standard ports, network packets do not reach our TCP server. Thus we modified the tool *wine* so that ports and destination addresses are rewritten on the fly triggered by an environment variable. This change was the first change of the scope of *wine*, because the purpose of *wine* is not to detour socket connections. The reader might notice that the goal of connection detours could also be done on a routing level or by library *detours* even on closed source software. At that time of study, our choice was the easiest to implement.

5.5 Monitoring system function calls

Closed source software like *detours* can be used for doing system call monitoring in a Microsoft Windows operating system. In our sandbox based on free software only an environment variable must be set in order to profit from a debugging feature implemented in *wine*. All executed functions belonging to *wine* or related libraries are printed on standard error. This feature can be discovered by reading the source code of *wine* or by reading the documentation provided by the developer community of *wine*. As already discussed in section 3.2 an algorithm should decide which function calls belong to the targeted monitored program and which ones belong to concurrent programs. A naive approach is to monitor function calls belonging to the memory area containing the machine instructions of the targeted monitored program. Unfortunately malicious programs often try to evade a monitoring process and they may generate machine instructions in other controlled processes or in dynamic allocated memory. Due to the fact that we studied the source code of the tool *wine*, we noticed that debug messages can be used to reconstruct a memory map, where functions are located.

By a formal exclusion mechanism we can decide which functions belong to the malicious program which belong to concurrent programs, which functions are internal ones [31]. We executed only one malicious program at once. Thus we have only to decide which function calls belong to *wine* and which ones to the malicious program.

In figure 3 we see that each function call has a return address. This is the address in memory where the program

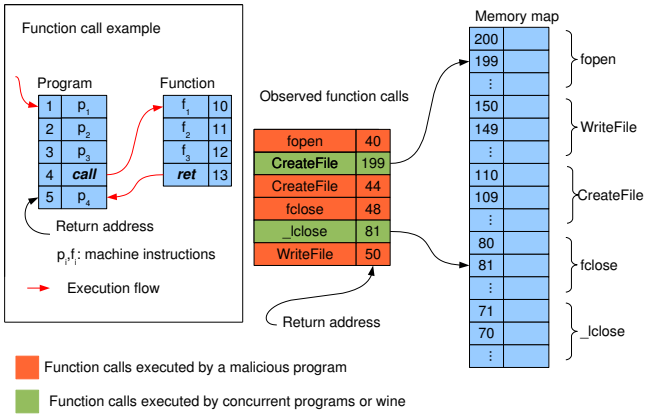

Figure 3. System function call filtering

continues its execution after the function call. In the debug messages these return addresses are included. Along with other debug messages we can reconstruct the memory layout.

5.5.1 Interactive sandbox

As described in section, 5.2, ANNE extracts high-level actions from malicious programs like other proprietary sandboxes. While reading a generated report from a sandbox containing actions performed by a malware sample, questions emerge like:

- If the malware sample is executed again, does the malware sample perform the same actions?
- What is the termination cause of the malware sample? Was it killed by the sandbox or were conditions not fulfilled for the execution, like missing files or registry keys.

These questions served as motivation to (i) understand the implementation details of the sandbox and (ii) to improve the monitoring process. To achieve this, access to the source code is needed in order to study it and modify it.

Due to the Halting Problem we were not able to find a master algorithm for solving all these questions. Thus we opted for a manual user interactive sandbox, shown in figure 4, serving as a tool for elaborating execution heuristics for malware monitoring. For this purpose we developed additional features in *wine*, like setting invisible break-points and memory dump facilities. The additional features were then used by a custom, also free, debugger called *fiw* [28] that we published at the security conference [30]. Moreover we developed additional features like disassembling

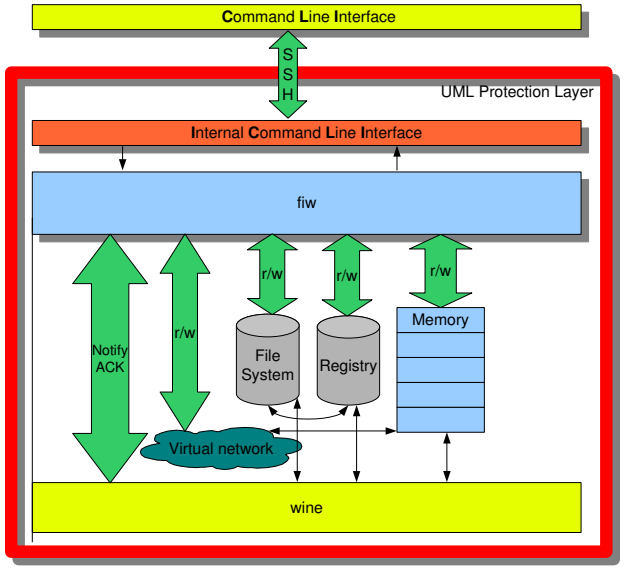

Figure 4. FIW architecture

memory, using free disassemblers, performing environmental changes, the interception of TCP communications. By environment changes we mean file changes and registry changes. The key concept is that each function call executed by *wine* must be acknowledged by the debugger. We also developed some automated debug actions, used for reaching the state of a malware sample in a quicker fashion. An example is a multi-threaded malware. One thread is scanning the network for the purpose of propagation and another thread implements a backdoor. If only the internals of the backdoor are studied the function calls related to the network scan can be acknowledged automatically.

6 Limits of FLOSS instrumentation

The freedom of studying the source code and the freedom of changing the source code of free analysis tools are great benefits in the context of malicious software analysis. However some freedoms can turn into limits.

Due to the fact that the source code is available for a free sandbox, malicious programs can be more easily instrumented to corrupt analysis results by exploiting implementation weaknesses. Despite giving malicious authors this freedom we prefer to keep the legal and technical capabilities for improving a sandbox. However it was shown that analysis results can be corrupted without studying the source code [26].

Free software is often developed by a single author doing the initial work to make the program usable. During the life time of the program, authors may abandon the development or make sporadic changes. In such case, the free software while being included in free software distribution is managed by each of them. This could lead to situation where the free software is forked and patch management is inconsistent among the various distribution [11].

A second challenge is the management of a project forks. In case the scope of a program is changed, the original author of the program does not care about the fork, due to the fact that the fork does not target their goals. They continue to modify their software and sometimes break patches of the fork. We experimented this with the tool *wine*. The *wine* project is a highly active project and our customized patches do not work on the latest version, after only a few weeks.

Fortunately the new possibilities offered by distributed version control (e.g. git) can mitigate these problems. The advantage of such version systems versus traditional ones, is that no central code repository is required. The developers of the main project have their repository and the developers of the fork have their own repository. The developers of the fork can then merge with the upstream repository and benefit from all the respective contributions.

7 Related work

Market share studies of free software in different IT application domains are presented in [33]. A quantitative measurement was done in order to compare free software with closed source software. Moreover it summarizes some benchmarks where free software is compared with closed source software. Security through obscurity is another approach but known to be flawed [15]. In this case, software or protocols are kept secret hoping to reduce the chance that a cracker attacks the software or protocol. Vulnerabilities in a product are often considered as a bad advertisement. Therefore many closed source software license agreements forbid public criticism [33].

In the context of security analysis of software, tools were developed to inspect the source code in a systematic way aiming to find possible vulnerabilities [32]. Access to the source code is a mandatory requirement for using such tools.

In the field of malware analysis closed source like *VMware*, and *IDA Pro* are frequently used, ready to use for malware analysis. Another quick solution is to resort to services, offered by third-parties, for doing this tedious task.

However malware researchers sometimes use free software in order to implement and integrate their innovation. A free decompiler for studying control flow graphs of disassemblers are used by [7]. Hybrid approaches, between closed source and free software , were proposed to examine closed source software. A free PC emulator was mod-

ified which runs a proprietary windows operating system designed specifically for malware analysis [4].

8 Future work in FLOSS

As long as authors of malicious software are motivated to continue to write malware samples the chances are high that they find countermeasures against monitoring techniques. Additional effort is required by malware analysts to find solutions which overcome these countermeasures. A direct consequence of this game is that a continuous effort is needed to improve malware sandboxes in order to avoid their future inefficiency. Free software helps to reduce development efforts due to the fact that more people can do the tasks, so everyone profits from the freedom F_1 and the freedom F_3. Moreover an empirical comparison can be done between the malware sandboxes. A prerequisite for this task is to solve legal and technical problems. Security codes are frequently used by the Norman Sandbox and the CWSandbox in order to hinder robots that submit malicious programs. Such a technique creates a strong limitation to generate a large data set of reports about malicious programs.

9 Conclusion

The analysis of malicious programs is sometimes challenging. We started by recalling freedoms of free software. We presented common applied methodologies and tools for closed source analysis with their limits. By quickly designing a software prototype, we sometimes noticed that a solution is worthy to be fully implemented or not. We had this experience with our idea to build a malware sandbox. The modification of free software reduces the amount and costs of the development time due to the fact that reuse of code is maximized. We managed to provide a proof of concept of a malware sandbox, built with free software having similar features to closed malware sandboxes. Moreover we were able to merge a debugger and a sandbox conceptually and implemented it by modifying free software and developed new free software [28, 27]. Closed source and free software suffer from hard theoretical limits. Closed source analysis tools often have some artifacts that are exploited by malicious programs in order to detect them. We noticed that free software is not guaranteed to be better than closed source software but the freedoms of free software enable a better camouflage by changing the source code. We preferred a sandbox based on free software where we can better evaluate the risk and accuracy of the sandbox by going through implementation details, having the legal and technical possibilities for performing improvements.

Acknowledgments

This work is partially funded by U-2010, an integrated research project of the 5th Call of the 6th European Research Frame Program. We want to thank Dr. Radu State and the members, of the SECAN LAB and LIASIT of the University of Luxembourg.

References

[1] D. Aucsmith. Tamper resistant software: An implementation. In *Proceedings of the First International Workshop on Information Hiding*, pages 317–333, London, UK, 1996. Springer-Verlag.

[2] J. Aycock. *Computer Viruses and Malware*. Springer, 2006.

[3] V. R. Basili, M. V. Zelkowitz, D. I. Sjøberg, P. Johnson, and A. J. Cowling. Protocols in the use of empirical software engineering artifacts. *Empirical Softw. Engg.*, 12(1):107–119, 2007.

[4] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research*, 2006.

[5] J. Beck. Using the cvs version management system in a software engineering course. *J. Comput. Small Coll.*, 20(6):57–65, 2005.

[6] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[7] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, 2007.

[8] CollabNet. Subversion. http://subversion.tigris.org/, 2000.

[9] A. Dulaunoy. Security and free software: Friends? In *Proceedings Hack.lu*, October 2006.

[10] Éric Filiol. *Computer Viruses: from theory to applications (Collection IRIS)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[11] K. Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc., 2005.

[12] http://www.gnu.org/. Last accessed: 18 September 2009.

[13] Hex-Rays. Ida pro. http://www.hex-rays.com/idapro/. Last accessed August 2008.

[14] G. Hunt and D. Brubacher. Detours: binary interception of win32 functions. In *WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[15] A. Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, 9, 1883.

[16] K. Korchinsky. Vmware fingerprinting counter measures. http://seclists.org/honeypots/2004/q1/0015.html. Last Accessed: August 2008.

[17] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM.

[18] Norman. Norman sandbox whitepaper. http://www.norman.com/documents/wp_sandbox.pdf, 2003.

[19] H. O'Dea. Trapping worms in a virtual net. In *Virus Bulletin Conference*, pages 176–186, 2004.

[20] A. J. O'Donnell. When malware attacks (anything but windows). *IEEE Security and Privacy*, 6(3):68–70, 2008.

[21] E. S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[22] M. Russinovich and B. Cogswell. http://technet.microsoft.com/en-us/sysinternals/bb896642.aspx, November 2006.

[23] M. Russinovich and B. Cogswell. http://technet.microsoft.com/en-us/sysinternals/bb896652.aspx, November 2006.

[24] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, 2002.

[25] L. Torvalds. GIT. http://git-scm.com/, 2005.

[26] A. Vidstrom. Evading the norman sandbox analyzer. http://www.ntsecurity.nu/onmymind/2007/2007-02-27.html. Last accessed: August 2008.

[27] G. Wagener. ANNE automated analysis and network emulation. http://git.quuxlabs.com/?p=anne/.git;a=summary.

[28] G. Wagener. FIW. http://git.quuxlabs.com/?p=fiw/.git;a=summary.

[29] G. Wagener, A. Dulaunoy, and T. Engel. Development and design of a process and a piece of software to analyze unknown software. Technical report, University of Luxembourg, 2006.

[30] G. Wagener, R. State, and A. Dulaunoy. Automated malware analysis. In *Proceedings of Hack.lu 2007*, October 2007.

[31] G. Wagener, R. State, and A. Dulaunoy. Malware behaviour analysis. *Journal in Computer Virology*, 2007.

[32] D. A. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder/.

[33] D. A. Wheeler. Why open source software / free software (OSS/FS)? look at the numbers! http://www.dwheeler.com/oss_fs_why.html, April 2007.

[34] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.