A study on practices against malware in free software projects

Ignacio Colomina, Joan Arnedo-Moreno, Robert Clarisó
Estudis d'Informàtica, Multimedia i Telecomunicació
Universitat Oberta de Catalunya
Barcelona, Spain
{icolomina,jarnedo,rclariso}@uoc.edu

***Abstract*—Many popular applications are developed using a free software model, through the collaborative effort of a community which makes the source code available for free. Unfortunately, malicious third parties may attempt to take advantage of this combination of popularity and openness by introducing software components that infect end-users who install the application. To reduce this security risk, several technical procedures and community management practices can be used during software development and distribution. This paper studies these procedures in the free source domain and evaluates their application in two widely-used open source projects, Symfony and Chromium.**

***Index Terms*—free software;open source; software development; security; malware;collaboration**

I. Introduction

Nowadays, many popular applications are developed under the Free Software banner, a movement that advocates that programs should be used, studied, modified, copied and re-distributed with almost no restrictions. In order to achieve this goal, such applications are developed as open source, usually through the collaborative effort of a developer community. Even though such communities may sometimes be conformed at its core by a tightly knit set of programmers, the very nature of free software allows any individual to selflessly contribute.

From a security standpoint, the free software development model offers a great advantage over its closed alternative [1], [2], as it allows any individual to freely inspect the code and look for bugs or vulnerabilities, so they can be quickly corrected by the developer community or the finder himself. The bigger the community, the greater the probability of finding and fixing errors in a timely manner. This is considered an advantage since it is a well known fact that security through obscurity, the approach often used by closed software, simply does not work at all [3].

Unfortunately, as far as security is concerned, there are three main scenarios where the freedom to study, modify and distribute source code becomes a liability. The first one appears when a malicious developer detects but does not disclose the vulnerability, hoping to exploit it at a later time. In the second scenario, a malicious developer submits a vulnerability or malicious component to the project code repository, hoping it goes unnoticed and becomes distributed in an official release. In the third scenario, a malicious party pretends to offer a copy or mirror of the original application, but offers malware instead. It is the developer community's responsibility to prevent these scenarios from ever happening, on the risk of losing its reputation.

Up to date, several dimensions of open source software development have been thoroughly studied, such as the development process of the most popular open source projects, e.g. Apache and Mozilla in [4], or the quality of the code being produced, e.g. [5] or [6]. However, the malware perspective in the context of open source has not been previously considered. In this paper, we present a study and evaluation on the procedures to avoid malware in the free software domain. Given the number of distinct free software applications available nowadays, two particular widely-used open source projects have been chosen for this study, Symfony and Chromium. This study is based on the case study research methodology in order to get results about the strength of its communities.

The remainder of the paper is organized as follows. First, Section II introduces the security risks in open source projects regarding malware and the related word. Then, Section III proposes several procedures that can be used to minimize the risk of malicious software appearing in the code base. Section IV presents two case studies of open software projects where the procedures are analyzed. Finally, Section VI draws conclusions from the results of the case studies.

II. Problem Description

In this paper, we study how to develop and distribute a free software project which is free from *malicious software*, i.e. malware. We consider the most abstract notion of malware, which can be defined [7] as "a set of instructions that run in your computer and make your system do something that an attacker wants it to do". This is a generic concept which covers a wide variety of threats, e.g. rootkits, trojans, ... However, we do not focus on the goals of this malware or the techniques it uses to operate, hide or infect systems. Instead, our aim is aiding in the prevention of them reaching the project code base and the end users.

Our problem description considers two different factors as far as malware propagation is concerned. First, we analyze the inherent vulnerabilities of free software projects because of its "free software" nature. Then, we study the role of the developer community in this process.

Other threats, such as compromising the server hosting the project source code to inject malware [8] are not considered in this work, as they may occur in any project regardless of the development model.

A. *Free software and its impact in security*

The original freedoms that define the notion of free software were established by the *Free Software Foundation*[1]:

1) The freedom to execute a program for any purpose.
2) The freedom to study how the program works and modify its functionalities.
3) The freedom to redistribute copies of the program.
4) The freedom to redistribute modified copies of the program.

These freedoms may offer to users, developers and companies some interesting opportunities in comparison to closed or private software does due to the fact that source code can be analyzed freely. In fact, freedoms 2 and 4 require source code access to be effective. The main advantages can be summarized as:

- Developers can benefit from the changes and improvements of other developers.
- Users can use (for free) quality software developed with the knowledge of many developers.
- Companies can implement business models which they were unavailable with private software.

Despite all these advantages, free software can open a door to malware, mainly through the open source code. On that regard Payne C analyzes the open source code feature in [2], stating that having many developers working in the same project can help to detect malware since there are a lot of *eyes guarding the code*. On the other hand, he also states that a malicious developer could study source code without dealing with binary files in order to insert malware, infect the code and then redistribute his copy to infect other people. In other words, an attacker would use the free software features as a channel to spread malware. Besides, there is another important aspect which is discussed both in [2] and [5]: that having a high number of developers working in the project does not mean that many qualified *eyes* are looking for vulnerabilities through source code, since many of the developers do not always have the necessary or required experience to do it.

B. *Free software communities and security*

Communities are the entities by which developers can collaborate in a free software project and communicate with other people involved in it. At the beginning of this movement, communities only made possible for developers to share code and work together. Nowadays, they act as organizations which manage and direct the collective effort of all collaborators. Some of the functions communities usually carry out are the following:

- **Roles**: Members of communities usually have defined roles which specify their privileges and responsibilities.
- **Access control**: Source code is usually stored on distributed repositories where users have to login before accessing it.
- **Protocols and promotion**: Communities usually include protocols by which developers and other members can promote into new and more important roles.
- **Authority**: Many times, the group of most committed contributors (*Core Team*) are in charge of making decisions about the project. Of course, other members can express their opinion.

[1]http://www.fsf.org

There are many factors which could explain why communities have evolved into these complex organizations. However, the most important reason is security. Section II-A explained that free software features may help attackers to study source code and infect it without having to deal with binary files. That is the reason why communities become organized, and especially their security features become really important.

In order to protect access to source code and monitor who reads and modifies it, communities must be equipped with technical and social mechanisms (roles, security and the other ones mentioned in the last paragraph) which act as a deterrent against attackers: *The stronger the communities, the fewer the attackers who will be able to infect them*. Both technical and social mechanisms are really important since one of them cannot live without the other. For instance, if a community has good network security but poor member promotion, a member could hide his malicious intentions and hack the project from within.

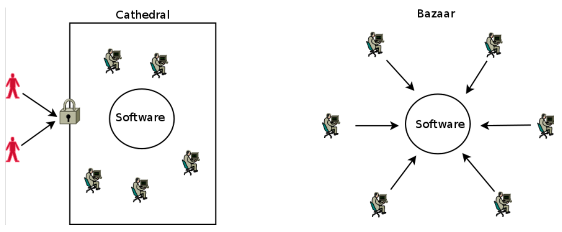

Fig. 1. The Cathedral and the Bazaar.

Eric Raymond's simile in [9] about The Cathedral and the Bazaar is a great example to understand how communities organize themselves. Figure 1 shows the Cathedral (a metaphor of closed source projects) as an entity where only members in it can collaborate in the project and external people have no access at all. On the other hand, the Bazaar (a metaphor for open source) acts as an entity where anybody who wants to collaborate is welcome. Nevertheless, current communities get the features both of the Cathedral and the Bazaar to make organizations that improve security and control and, at the same time, allow developers around the world to join the project by following a set of rules before starting to collaborate (Figure 2).

In this direction, [10] considers a community chart as an onion model. Onion layers which are next to the onion nucleus represent high responsibility roles and the other ones which are far from the nucleus represent low responsibility ones.

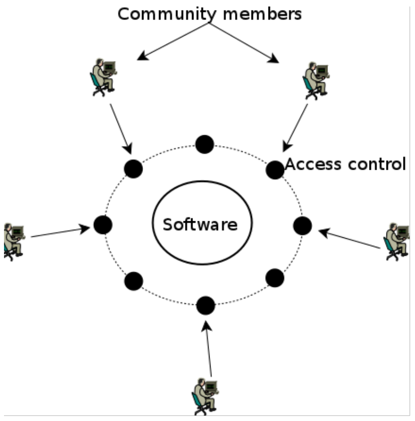

Fig. 2. Protected but accessible community.

That is the way which many communities use to make a promotion protocol where members can promote from remote layers to nearby layers. With this model, new developers and users can join communities (following the features of the bazaar) but only members who prove their professionalism get more responsibilities and can become Core team members (following the features of the cathedral).

Figure 3 shows an example of such an onion model community with five layers. New members join the community in the outermost layer, which grant them minor privileges within the project. As the member gathers trust from the community through useful contributions, it becomes promoted to the inner layers, which grant him additional privileges and a higher degree of trust in his actions. In the end, the committer becomes an experienced member of the community and may join the inner layer, becoming a Core Team member.

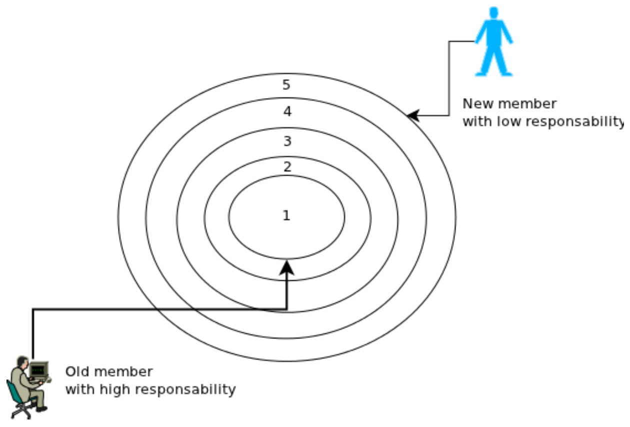

Fig. 3. Onion model for a project community

III. Procedures and best practices

Under a free software model, it is very important to have a set of procedures which can help to build strong communities and provide trust. This research is based on a list of procedures in order to evaluate the security of the two chosen communities (Symfony and Chromium). These procedures are important because they provide an easy way to know how secure a community is. This section first introduces how the procedures have been chosen and then describes them in detail.

A. Selection of procedures

The procedures proposed in this work have been inspired by the safeguards elements of the MAGERIT v2 standard for risk analysis, management and control (ISO1779) [11]. The categories for these safeguards are the following:

- Access control and privilege management.
- Organizational security.
- Action log.

Procedures are classified according to a level of importance: *normal* or *high*. This classification captures the risk of the absence or a failure in the implementation of the procedure. That is, normal procedures improve the degree of security but may not be as severe or as easy to exploit as high importance procedures.

B. Procedure description

The list of our proposed procedures and their descriptions are listed in Table I.

IV. Case studies

The use of case studies to make this research can help to know how much strong communities are by analyzing whether they carry out the procedures or not. As it is mentioned in II-B: *The stronger the communities the fewer the attackers who will be able to infect them.* The main features of this research are the following:

- Case studies will be exploratory studies.
- The instance of each case study will be a community of a free software project.
- The procedures defined in this research will be used as a conceptual framework in order to generalize these case studies to other projects.
- The source of information for analysing communities will be the Internet, and more precisely, the project web portal.

The chosen projects for this research are the following:

Symfony: A PHP framework which contains classes and libraries to develop web applications following the Model-View-Controller paradigm.

- Distribution license: MIT
- Project web page: http://www.symfony.com/

Chromium: An Internet browser which main aim is to be fast, secure and stable.

- Distribution license: The Chromium part developed by Google was released with BSD license and the other parts were released with several open source licenses such as MIT, LGPL, Ms-PL or the combined license MPL/GPL/LGPL
- Project web page: http://www.chromium.org/Home

Procedure	Description	Risk level
	Access control and privilege management	
P.1	A source code repository is used to keep track of versions. The repository is either centralized or, if it is distributed, it has a master branch used by all developers	High
P.2	A group of members controls who accesses the repository and their actions	High
P.3	There is a protocol to grant access to the repository managed by the community Core Team	High
P.4	The Core Team decides which changes are included in a new release	High
P.5	There are security mechanisms which guarantee the integrity of downloadable packages and source code	High
	Action log	
P.6	Developers have to follow code conventions	Normal
P.7	Procedures to control software quality (e.g. unit testing) are in place	High
P.8	There is a ticket software to record who changes source code and why	High
P.9	There is a procedure to revise new functionalities and changes made by other developer members	High
P.9.1	The review procedure is performed by members of the Core Team	High
P.9.2	The project includes a different branch for unrevised code before it is merged with the master branch	High
P.9.3	The procedure includes a connection between the ticket software and the P.9.1 review	Normal
P.10	A procedure is used in order to systematically produce a release	High
P.11	There is a system to generate version names which identifies their aim (A, A.B, A.B.C)	Normal
	Organizational security	
P.12	Community hierarchy follows an onion model	High

TABLE I
LIST OF PROPOSED SECURITY PROCEDURES FOR FREE SOFTWARE PROJECTS

These projects were selected because they are interesting, with a wide user base both among individuals and industry and have not been previously studied from a scientific perspective.

A. Procedure assessment criteria

The assessment of procedures was carried out by assigning a value for each procedure, depending on whether the community properly performs a it or not. Fortunately, free software projects provide guidelines to new contributors, with information about the organization of the community, the code contribution process, member privileges, etc. We used this information to assess each procedure in Symfony and Chromium.

After each procedure was properly assessed, it was evaluated according to the following criteria:

- Procedure has not been found or the community does not apply the procedure. Value: *Not satisfied*
- Procedure has been found but not properly applied by the community. Value: *Partially satisfied*
- Procedure has been found and the community correctly implements it. Value: *Satisfied*

B. Procedures assessment results

Table II summarizes the procedure evaluation results for each community. Unsurprisingly, since the projects under study are widely used and being actively developed, both of them satisfy most procedures. However, it is interesting to notice that, even for these high-profile projects, not all procedures are fully accomplished. Table III identifies the issues where some procedures failed to be satisfied.

C. Evaluation of communities

Each non satisfied procedure reduces the effort required by an attacker to get its malware distributed inside the free software project. Even though all procedures are equally important, as each addresses just a different kind of threat, it may be important to have an overview on the degree of security in a free software project. For example, this could be useful when comparing different free software projects, e.g. to select the most suitable project to use among several candidates.

Procedure	Symfony	Chromium
P.1	Satisfied	Satisfied
P.2	Satisfied	Satisfied
P.3	Partially satisfied	Satisfied
P.4	Satisfied	Satisfied
P.5	Partially satisfied	Partially satisfied
P.6	Satisfied	Satisfied
P.7	Partially satisfied	Satisfied
P.8	Satisfied	Satisfied
P.9.1	Satisfied	Satisfied
P.9.2	Satisfied	Satisfied
P.9.3	Satisfied	Satisfied
P.10	Partially satisfied	Satisfied
P.11	Satisfied	Satisfied
P.12	Satisfied	Satisfied

TABLE II
PROJECT ASSESSMENT SUMMARY

To this end, we propose a metric to approximately measure the degree of security against malware in a free software project. This metric is defined from the set of satisfied procedures in the following way.

- Satisfied procedures add 1 point.
- Partially satisfied procedures add 0.5 points.
- Non satisfied procedures do not add any points.

This metric ranges from 0 (worst) to 14 (optimal). To provide a qualitative view of this information, we may consider qualifying them according to the following intervals:

- Optimal strength: 14 points
- High strength: [11 - 14) points
- Medium strength: [8 - 11) points
- Low strength: [6 - 8) points
- Critical strength: [0 - 6) points

Project	Procedure	Issue
Symfony	P.3	No formal procedure to control access to the software repository.
	P.5	No checksum available when downloading a bundle (ZIP, ...)
	P.7	Insufficient quality control measures (only functional and unit tests required).
	P.10	No release calendar planned for new versions.
Chromium	P.5	No checksum available when downloading a bundle (ZIP, ...).

TABLE III
SUMMARY OF ISSUES FOUND IN SYMFONY AND CHROMIUM

Using the previously mentioned rules, the metric for each project is as follows:

Symfony:

- Partially satisfied procedures score: 4 * 0.5 = 2 points
- Satisfied procedures score: 10 * 1 = 10 points
- Total score: 10 + 2 = 12 points (**High strength**)

Chromium:

- Partially satisfied procedures score: 1 * 0.5 = 0.5 points
- Satisfied procedures score: 13 * 1 = 13 points
- Total score: 13 + 0.5 = 13.5 points (**High strength**)

V. DISCUSSION OF RESULTS

After knowing the results and qualification for each project, it is the time to further discuss how an attacker could take advantage of a procedure not being satisfied. each procedure which is not completely satisfied is individually discussed.

A. Symfony

Procedure P.3 (Partially satisfied): Having a procedure which acts as a way to become a committer can help a community to protect itself against attackers. Although the community Core Team revises all committed code there can be committers with malicious aims who try to introduce malware hidden in a new functionality or code change. By establishing this procedure, future malicious committers can be discovered before they are able to login into the software repository.

Procedure P.5 (Partially satisfied) Symfony users can download the software in a bundle (zip, tar, tar.gz, etc.). However, this package does not include a verification checksum such as SHA-1 [12] in order to check its integrity. Users downloading the software through unofficial means (e.g. a peer-to-peer site or a fake mirror) may receive an altered bundle infected with malware.

Procedure P.7 (Partially satisfied) Although developers have to build a functional and unit test to each code change they do, a malicious committer could add malware through a test. In order to avoid it, a community should have its own testing software and all committers should use it to build tests.

Procedure P.10 (Partially satisfied) Symfony does not have an release calendar for future releases. An attacker could benefit from this absence by faking a new release, urging users to download a malware-infected version of the project source.

B. Chromium

Procedure P.5 (Partially satisfied) Just like in Symfony, Chromium bundles do not include a verification checksum. Therefore, users have no way of checking whether the software they downloaded through unofficial channels has been tampered.

C. Summary

In the projects under study, the potential vulnerabilities could be exploited to release a fake version of the project which can be used to spread malware. In particular, the omission of procedure P.5 (integrity verification) can be used, for instance, to seed a "mirror" of the project with an infected version of a release.

This issue is quite easy to solve and it is already addressed by a large group of free software projects, which provide integrity verification checksums for all their package downloads. Some examples of projects which apply this best practice are: the Apache HTTP server, the MySQL database server, the SquirrelMail webmail client, the deployment tool Apache Ant, the print system Cups, etc.

VI. CONCLUSION

Malware avoidance in open source software is still an open problem, as it can be shown by several recent outbreaks such as the mentioned in [13] and [14]. In fact, the projects in our case study (Symfony and Chromium) illustrate that even popular open source projects lack some simple safeguards that protect against the introduction of malware.

Several factors contribute to the complexity of this problem. First, there is no single "silver bullet", a combination of procedures must be used to address all potential risks. Second, solutions require both technical measures (e.g. authentication required for commits to the code base, security of the software repository) and community management measures (e.g. layered community with different roles and levels of privilege). And third, some procedures, like reviewing all software patches submitted to the project, require a continuous effort from the developer community.

As a future work, the following approaches should be taken into account:

- From this research, design a procedure list which can be standardized and used as a guideline to make strong communities.
- To make an analysis about economic losses which could generate a free software project because of not achieving standardized procedures. For instance, by using eigenvalues and eigenvectors.

Acknowledgements

The author Ignacio Colomina would like to thank to M^a Soledad Ferri Sellés for her advice in English grammar and vocabulary.

References

[1] J.-H. Hoepman and B. Jacobs, "Increased security through open source," *Commun. ACM*, vol. 50, no. 1, pp. 79–83, jan 2007. [Online]. Available: http://doi.acm.org/10.1145/1188913.1188921

[2] C. Payne, "On the security of open source software," *Information Systems Journal*, vol. 12, no. 1, pp. 61–78, 2002. [Online]. Available: http://dx.doi.org/10.1046/j.1365-2575.2002.00118.x

[3] A. Kerckhoffs, "La cryptographie militaire," *Journal des sciences militaires*, vol. IX, 1983.

[4] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, 2002.

[5] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, 2002. [Online]. Available: http://dx.doi.org/10.1046/j.1365-2575.2002.00117.x

[6] L. Zhao and S. Elbaum, "Quality assurance under the open source development model," *Journal of Systems and Software*, vol. 66, no. 1, pp. 65 – 75, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412120200064X

[7] L. Z. Ed Skoudis, *Malware. Figthing malicious code*. Prentice Hall, 2004.

[8] S. Anthony. (2012, 03) Github hacked, millions of projects at risk of being modified or deleted. [Online]. Available: http://www.extremetech.com/computing/120981-github-hacked-millions-of-projects-at-risk-of-being-modified-or-deleted

[9] E. S. Raymond, *The Cathedral and the Bazaar*. O'Reilly, 2001.

[10] M. Aberdour, "Achieving quality in open source software," *IEEE Software*, vol. 24, no. 1, pp. 58–64, 2007.

[11] M. of Civil Services of Spain. EAR / PILAR - entorno de análisis de riesgos. [Online]. Available: https://www.ccn-cert.cni.es/index.php?option=com_wrapper&view=wrapper&Itemid=187&lang=es

[12] NIST, "FIPS PUB 180-1: Secure hash standard," 1995, http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[13] D. Goodin. (2011, 08) Kernel.org linux repository rooted in hack attack. [Online]. Available: http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/

[14] M. Pithia. (2011, 10) Is your website accused of phising? Phishing sites on the rise due to web application vulnerabilities. [Online]. Available: http://info.brandprotect.com/Blog/bid/64544