Towards a unified software attack model to assess software protections

Towards a Unified Software Attack Model to Assess Software Protections

Cataldo Basile
Politecnico di Torino, Torino, Italy
cataldo.basile@polito.it

Mariano Ceccato
Fondazione Bruno Kessler, Trento, Italy
ceccato@fbk.eu

***Abstract*—Attackers can tamper with programs to break usage conditions. Different software protection techniques have been proposed to limit the possibility of tampering. Some of them just limit the possibility to understand the (binary) code, others react more actively when a change attempt is detected. However, the validation of the software protection techniques has been always conducted without taking into consideration a unified process adopted by attackers to tamper with programs.**

In this paper we present an extension of the mini-cycle of change, initially proposed to model the process of changing program for maintenance, to describe the process faced by an attacker to defeat software protections. This paper also shows how this new model should support a developer when considering what are the most appropriate protections to deploy.

***Index Terms*—Program comprehension, Software protection, Software security.**

I. Introduction

Encryption and firewalls are established solutions, largely adopted to block remote attackers (also know as Man-In-The-Middle attackers) who try to intercept and tamper with the communication to steal sensitive data or to break into software systems. However, these classic approaches do not help defending software systems in a context where the attacker is the end user itself, threat known as Man-At-The-End attack (MATE) [1]. MATE regards those client applications that are required to run under strict usage conditions, that could be violated on tampered clients. For example, client applications for media conditioned-access (e.g., pay-per-view digital TV) could be tampered with to access the service in a way that was not intended by the service provider (e.g., paying a reduced fee), and on-line game providers should prevent cheating to ensure a fair competition.

So far, many software protection techniques have been proposed with the objective of defending code from Man-At-The-End attacks, some of them are more passive while others react actively to attacks. Examples of passive protection techniques are *code obfuscation* [2] that prevents code comprehension by making code more hard to understand while preserving semantics, and *data hiding* [3] that aims at hiding sensitive data (e.g., cryptographic keys) within the program code. More active protections that react when a modification is detected are based on tamper-detection with *self-checkers* [4] and *protections update* [5]. *Remote attestation* [6] is also an active protection, it is used to detect and react to attacks by disconnecting tampered clients. Proofs about program integrity or other interesting properties are generated on the client and sent a remote verifier. Other techniques, e.g., *time bombs* or *graceful degradation* [7], can be used to make protected programs unusable if modifications are detected or if programs are used with no Internet connection for long periods. *Program splitting* [8] is another approach to make a tampered program useless, the original application is split into two parts so that the sensitive portion is executed in a trusted environment (e.g., a remote server). Finally, *Code replacement/update* [9] is an option to give the attacker a limited amount of time to succeed before a new version of the client is released.

Current solutions for software protection are based on the intuitive (sometimes implicit) assumption that, to be tampered with, a program needs to be understood and changed. Even if this intuition is correct, software protections miss a unified view on the process adopted by attackers to hack around code. In fact, every protection focuses on a single sub-goal towards the attack, without paying attention to the overall process and the collateral activities required to port an attack. For example, code obfuscation may not be effective in those contexts where understanding the whole program is not required, but a local knowledge is sufficient to locate the point of the program to change (e.g., the routine for license checking).

What is the new idea? With this paper we mean to raise the awareness of the program understanding community on this emerging problem, the need to study program understanding in the context of attacks to software protections. Moreover, we present the *mini-cycle of attack*, a preliminary version of a model intended to describe the process adopted by an attacker to elaborate an attack against software protection.

Why is this idea new? So far, research in program understanding was mainly devoted to study the challenges emerging from development, maintenance and (legitimate) reverse engineering of programs, while, to the best of our knowledge, topics related to the understanding of protected programs for malicious tampering are quite overlooked.

In fact the evaluation of software protections was mainly conducted by using code metrics [2], [10], [11], [12] or by measuring how long a tool takes to automatically remove a specific protection [13]. Theoretical validation could be done only for a limited class of protections [14]. Empirical validation involving human participants is a more reliable measurement, but it is very expensive and it was adopted only very seldom [15].

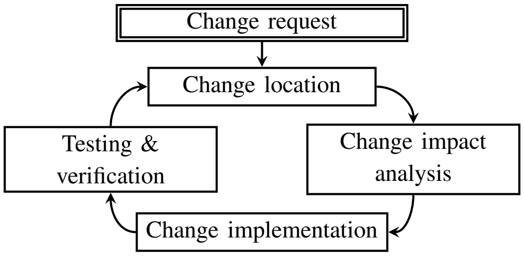

Fig. 1. Mini-cycle of change.

Why is this idea relevant? The immediate benefits of a unified model of the attack process are many. For instance, software protections approaches are rarely used alone, often different approaches are composed in the hope to guarantee a higher level of protection. However, without a unified vision on the attack process, the *understanding on the interaction of different protections* can be just intuitive, and the composition might be not effective. As a matter of fact, the trivial strategy *"the more protections, the more secure"* fails when a poorly designed composition is adopted, and a protection leaks the weak points of the other protection.

Additionally, empirical investigation and user studies [15] have been conducted to measure and compare the level of protection offered by distinct approaches. Despite very informative, these studies are very expensive to conduct. A comprehensive model of what are the challenging tasks faced by attackers is of fundamental importance to properly design the most important experiments to conduct. Identifying the crucial dimensions to investigate with higher priority would help in reducing the time wasted in experimenting on marginal aspects. For example, there is a huge number of potential combination of protection techniques and it would be too expensive to investigate all of them. A careful planning is required in order for the research community to be successful.

What is the most relevant paper? This paper builds on top of the *mini-cycle of change*, proposed by Rajlich [16] to describe the process adopted by software developers to perform maintenance tasks. The mini-cycle of change by Rajlich is revised and adapted to describe malicious software tampering, including the challenges faced by an attacker to defeat potential software protection techniques.

What is the expected feedback from the forum? With this paper we intend to initiate a discussion on the role of program understanding for software tampering and on what are the fundamental differences (and common aspects) with respect to program understanding for software development. Moreover, we expect comments, critics and suggestions on the mini-cycle of attack, on how to extend it and improve it. We also mean to discuss if the mini cycle of attack represents a valid support to those programmers who need to design defense strategies to protect against MATE attacks.

II. Background: The Mini-Cycle of Change

The first intuition of the mini-cycle of change was proposed in 1978 by Yau et al. [17] to model the process adopted by a programmer to complete a change task. This cycle was based on the assumption that a change often causes inconsistencies in the rest of the application code. They are "ripple effects" that make the application incompatible with the change, so they need to be taken into account and fixed in an iterative way. This first intuition was later refined and completed by Rajlic [16]. According to Rajlic, the software change corresponds to the iterative process summarized in Figure 1. It consists of:

1) **Change request:** The reason for the code modification initiates the change process. It may come from customers who need a new functionality or from users who report a defect;
2) **Change location:** The software is inspected (either manually or using tools) to acquire the minimal amount of information required to formulate a hypothesis on where to apply the change;
3) **Change impact analysis:** Side effects of the main change are identified throughout the application code (the *ripple effects*) in order to have the complete picture of the maintenance activity;
4) **Change implementation:** Once the change task is known and understood by the developer, the actual change can be performed;
5) **Testing and verification:** The changed program is tested to check whether the implemented corrections satisfy the initial change request.

If the verification highlights a problem, the mini-cycle is reiterated. This problem becomes the new *change request* for the subsequent iteration. The cycle is iterated until all the problems are solved and the initial change request can be considered fully satisfied. Possibly, when the change task is over, software documentation is updated to record changes.

Differently from those models that consider complete software life-cycle (e.g., the waterfall development model), the mini-cycle is meant to model just modifications. It captures the process initiated by the developer to respond as fast as possible to requests for change.

III. The Mini-Cycle of Attack

Although software maintenance and malicious software tampering have fundamental different objectives, we claim that they share very similar operations. In a software maintenance scenario, the modification of the software is motivated by a bug report or by a request for a new feature. While, in case of attacks, the modification is due to the attacker's intention to misuse an application or to "steal" developer's intellectual property. However, even with different rationales, both of these activities involve a cognitive intensive effort to map the change goal to the code, to consider the side effects of this change and to evaluate whether the change (or the attack) is successful or if further actions must be planned. Therefore, our claim is that maintenance changes and attacks can be described using a similar model. The mini-cycle of change is a good starting point to describe attacks to software, but it needs to be adapted and take into account the specificity of an attack context.

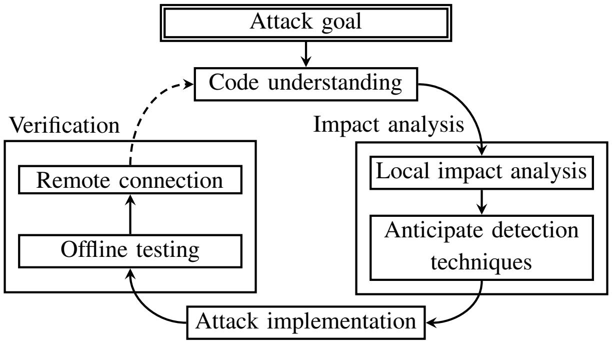

Fig. 2. Mini-cycle of attack.

The objectives of the attacker are (i) to mount an attack as fast as possible and (ii) to reduce or avoid the risk of being detected by software protections. Thus, the original mini-cycle of change, proposed for software maintenance, needs to be refined (see Figure 2). It consists of the following phases:

1) **Attack goal:** This is the final goal of the attacker and it depends on the asset that the software is supposed to preserve (e.g., a secret key, a license checking routine);
2) **Code understanding**: The change location step is renamed in *code understanding*, because the attacker is not necessarily supposed to change the original code. In fact, depending on the attack goal, a successful attack may be limited to steal data or code from the application (e.g., locate a function and use it out of intended context). While change location (mini-cycle of change) is performed on known/documented/commented source code, code understanding (mini-cycle of attack) is performed on reverse-engineered, probably obfuscated, code. In any case, this step represents the cognitive intensive activity, needed to plan the next steps.
3) **Impact analysis:** Impact analysis is split in two sub steps to remark that an attacker must also consider active protections (e.g., anti-tampering or remote attestation):
    - **Local impact analysis:** This step includes the analysis of those side effects whose consequences are just local. It is similar to the mini-cycle of change;
    - **Anticipating detection techniques**: Other side effects could have consequences that are not meaningful at the functional level. However, these changes could be noticed by local protection techniques or by a remote entity that may initiate defensive strategies (e.g., disconnect the tampered client). In order to defeat protection mechanisms, the attacker should elaborate a change that is undetectable.

    This step is more complex than the corresponding *change impact analysis* (mini-cycle of change), because of incomplete knowledge, e.g., the attackers does not know what side effects are monitored to reveal tampering.
4) **Attack implementation:** The attack is applied as elaborated in the previous steps;
5) **Verification:** Verification is split in two sub-phases:
    - **Offline testing:** Debugging and testing can be used to verify that the attack is successful with respect to local resources, i.e., without involving any remote party;
    - **Remote connection:** In this case, the tampered application is required to interact with a remote component, as in the case of on-line games.

    The presence of a remote verifier is the main difference with respect to the mini-cycle of change. In fact, the remote verifier could black list detected attacks and prevent the "trial and error" strategy.

Additional iterations might be required if the attack is not successful after the first one. However, a second attempt might be impossible, if tampering is detected by a remote verifier.

The mini-cycle of change fits only those cases when new attacks need to be elaborated. This analysis does not fit the cases of automatic attacks, e.g., when existing cracks or patches allow the attacker to reach her/his goal automatically.

IV. Adoption of the Mini-Cycle of Attack

The mini-cycle of attack clearly identifies all the phases of an attack. It represents a valid support for a software developer when reasoning on which software protections to deploy and when judging on the trade-off between protection costs (e.g., time, memory, performance overhead) and expected benefits.

The effect of software protection techniques can be analyzed in the light of the mini-cycle of attack. Obfuscation, data hiding, and execution in trusted environment are protections that operate in the *code understanding* phase by making the code more hard to understand. Some variants of obfuscation (e.g., control flow flattening) partly protect *Local impact analysis*. Remote attestation and anti-tampering protect the *anticipate detective technique* phase, because they force the attacker to elaborate undetectable implementations. Code splitting, and time bombs also block or limit the *offline testing*. Execution in trusted environment and remote attestation are expected to protect the *remote connection* phase.

Moreover, the mini-cycle of attack can be used to design protections with special emphasis on the combination of elementary techniques. The conception of protections starts from the identification of the assets to protect.

For instance, let's assume that the defender needs to protect the intellectual property of a very efficient implementation of mathematical functions, because it required a significant investments of resources and it represents a competitive advantage over the competitors. The adoption of code obfuscation could protect in the *code understanding* phase and possibly in the *local impact analysis* phase by turning the (binary) code very hard to understand and to reverse engineer. At this stage, adding more protections to the same two steps could bring limited advantages. It might be more appropriate to try and protect other steps that are relevant to the defender goal, but still not adequately protected, such as making impossible to do *offline testing*. Additionally, the defender should limit the possibility for the attacker to extract a portion of code and reuse it in a different context, such as in a product by a competitor. This could be done by integrating software guards, an active protection approach where runtime verification of selected program properties makes the mathematic routine malfunction (e.g., produce wrong results) when runtime verifications do not pass. This could be easily achieved by including values coming from the verification in the mathematical computation, at the price of a slightly slower performance. If the residual risk or the performance degradation is not acceptable, the most sensitive parts of the code could be executed remotely in a trusted environment. This in turn will also help protecting from extensive offline testing or trial and error approaches. The objective of protecting this particular example is to turn the attack task so expensive that it becomes uneconomical for the attacker to try and steal the software asset. Probably, an attacker may give up and devote available resources to the development of a brand new mathematical library.

The steps to focus on, of course, depend on the attack that the defender means to prevent. Let's assume that it is of critical importance to protect a program against misuse, for example an on-line game with a considerably large client-side installation. In this context, the players community is very important and may decide the success or the failure of a new game, so cheating should be prevented to preserve the game reputation. Since this asset is different, the protections to deploy will be also different. In fact, the most important attack step to block is probably the *verification*, in particular in the *remote connection* sub-step, to be able to detect tampered clients and disconnect them from the central game server. This could be achieved with remote attestation techniques that should detect tampering. These techniques are meant to make hard for the attacker to circumvent remote controls during the *anticipate detective techniques* phase. Of course, other steps could be also protected with different protection strategies, but protecting other parts of the mini-cycle of attack is important as long as it is of help to discover tampered clients.

V. Conclusions and Work-in-Progress

In this paper we extended and adapted the mini-cycle of change, to model the behavior of an attacker who intends to defeat software protections and tamper with programs. We proposed the mini-cycle of attack and we showed how it can be used to support software developers in planning what protection(s) to deploy and how to compose them.

This preliminary version of mini-cycle of attack needs to be validated and refined, possibly by conducting a systematic literature review on how software protection have been validated and measured (both theoretically and empirically). We plan to map existing works to a common framework, i.e., the mini-cycle of attack. This mapping will make it possible to define a research agenda of the user studies to conduct, to quantify and compare protection techniques.

Among our future works, we intend to rely on the mini-cycle of attack to build a decision support system, a tool to support software developers in identifying the connections between protections and the assets to protect, and to guide the evaluation of the trade-off between protection benefits and protection costs (e.g., performance overhead). Moreover, we will evaluate if the mini-cycle of attack can be used to perform risk analysis and assessment, to quantify the security risk.

Acknowledgment

Authors would like to thank professor Yoram Ofek, who recently passed away, for having inspired this paper.

References

[1] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, "Guest editors' introduction: Software protection," *IEEE Software*, vol. 28, no. 2, pp. 24–27, March 2011.

[2] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. of Computer Science, The Univ. of Auckland, Technical Report 148, 1997.

[3] W. Michiels and P. Gorissen, "Mechanism for software tamper resistance: an application of white-box cryptography," in *Proceedings of the 2007 ACM workshop on Digital Rights Management*, ser. DRM '07. New York, NY, USA: ACM, 2007, pp. 82–89.

[4] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *ACM Workshop on Security and Privacy in Digital Rights Management*. ACM, 2001.

[5] R. Scandariato, Y. Ofek, P. Falcarin, and M. Baldi, "Application-oriented trust in distributed computing," in *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*. IEEE, 2008, pp. 434–439.

[6] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *Int. J. Inf. Secur.*, vol. 10, no. 2, pp. 63–81, Jun. 2011.

[7] G. Tan, Y. Chen, and M. Jakubowski, "Delayed and controlled failures in tamper-resistant software," in *Information Hiding*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4437, pp. 216–231.

[8] M. Ceccato, M. Preda, J. Nagra, C. Collberg, and P. Tonella, "Trading-off security and performance in barrier slicing for remote software entrusting," *Automated Software Engineering*, vol. 16, pp. 235–261, 2009.

[9] M. Ceccato, M. D. Preda, J. Nagra, C. Collberg, and P. Tonella, "Barrier slicing for remote software trusting," in *Proc. of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE Computer Society, Sept. 30 2007-Oct. 1 2007, pp. 27–36.

[10] H. Goto, M. Mambo, K. Matsumura, and H. Shizuya, "An approach to the objective and quantitative evaluation of tamper-resistant software," in *Third Int. Workshop on Information Security (ISW2000)*. Springer, 2000, pp. 82–96.

[11] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel, "Program obfuscation: a quantitative approach," in *QoP '07: Proc. of the 2007 ACM Workshop on Quality of protection*. New York, NY, USA: ACM, 2007, pp. 15–20.

[12] A. Capiluppi, P. Falcarin, and C. Boldyreff, "Code defactoring: Evaluating the effectiveness of java obfuscations," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, Oct., pp. 71–80.

[13] S. Udupa, S. Debray, and M. Madou, "Deobfuscation: reverse engineering obfuscated code," *Reverse Engineering, 12th Working Conference on*, Nov. 2005.

[14] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Advances in Cryptology (CRYPTO 2001)*, J. Kilian, Ed. Springer Berlin Heidelberg, 2001, vol. 2139, pp. 1–18.

[15] M. Ceccato, M. Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques," *Empirical Software Engineering*, pp. 1–35, 2013.

[16] V. Rajlich, "Software change and evolution," in *SOFSEM99: Theory and Practice of Informatics*, ser. Lecture Notes in Computer Science, J. Pavelka, G. Tel, and M. Bartoek, Eds. Springer Berlin Heidelberg, 1999, vol. 1725, pp. 189–202.

[17] S. S. Yau, J. S. Colofello, and T. MacGregor, "Ripple effect analysis of software maintenance," in *Proc. COMPSAC*. IEEE Computer Society Press, 1978, pp. 60–65.