Security Hardening of Open Source Software

Azzam Mourad, Marc-André Laverdière and Mourad Debbabi
Computer Security Laboratory, CIISE,
Concordia University, Montreal (QC), Canada
{mourad,ma_laver,debbabi}@ciise.concordia.ca [*]

Abstract

In this paper, we define the concept of software security hardening, which will allow the developers and maintainers to deploy and harden security features and remedy present vulnerabilities and threats into existing open source software. We also propose a classification of the different levels at which the hardening can be applied and a methodology for hardening of high level security into applications based on a well-defined security ontology. In addition to this contribution, we elaborate the methods for hardening security vulnerabilities found in C according to the classification we propose.

1. Motivations & Background

In todays computing world, security takes an increasingly predominant role. The industry is facing challenges in public confidence at the discovery of vulnerabilities, and customers are expecting security to be delivered out of the box, even on programs that were not designed with security in mind. Software maintainers must face the challenge, today, to improve the security of their programs, and are often under-equipped to do so. Some countries are taking advantage of open source software for their production systems as the availability of the source code facilitates their validation and answers their need for trustworthy programs. Open source software are typically implemented using the C programming language [1] and, as such, it is necessary to investigate the security issues related to C.

This paper provides the first academic attempt at security hardening, and demonstrates its applicability on the C language. We refer to the process of integrating security into existing software as security hardening, as this practice often refers to modifying the program in a way that makes it more resistant against attacks. In the current context, it becomes increasingly important to provide tools to maintainers that will facilitate and accelerate the security hardening process, increasing the effectiveness of the effort and lowering the resources required to do so.

The rest of this paper is organized as follows. In section 2, we introduce the important contributions on the field of secure programming. Afterwards, in section 3, we define security hardening and propose a classification of its different levels. In section 4, we present a methodology for high level security hardening based on a well-defined security ontology. In section 5, we elaborate the methods of hardening against vulnerabilities related to C programs, structured according to our classification. Finally, we offer concluding remarks in section 6.

2. Related work

The topic of hardening is mostly known at the level of the operating system and network configuration (i.e. Bastille Linux [1]). Our approach constitutes an organized framework for methodologies for the improvement of security at all levels of the system. As such, the terminology of "hardening" that we propose is not the same as for operating system hardening. Currently, security solutions can be found in secure coding books, in programmer/reviewer checklists, and in the mind of many experts. This help does not offer practical support on how to deal with legacy code and how to harden security into existing software.

On the topic of secure programming of C programs, developers are offered a good selection of useful and highly relevant material. One of the newest and most useful additions is from [9], which offers in-depth explanations on the nature of all known low-level security vulnerabilities in C and C++. Its treatment of integer overflows is the best we found in the literature.

Another common reference is from Microsoft [5], and includes all the basic security problems and solutions, as

[*]This research is the result of a fruitful collaboration between the computer security Laboratory of Concordia University, DRDC-Valcartier National Defense Canada and Bell Canada thanks to a grant under the NSERC/DND Research Partnership Program.

[1]according to SourceForge.net statistics, 26.22% of open source projects at "Production" and "Mature" levels are written in C [7]

well as code fragments of functions allowing to safely implement certain operations (such as safe memory wiping). The authors also describe high-level security issues, threat modeling, access control, etc.

Slides from Bishop(such as [3]) provide a comprehensive view on information assurance, as well as security vulnerabilities in C. He is one of the few who cover in depth environmental issues, used in some advanced exploits. In addition, he provides some hints and practices to solve some existing security issues.

In the literature, the reference that is the closest in comprehensibility to our work comes from Wheeler [10]. In the material he published online, he covers operating system security, safe temporary files, cryptography, multiple operating platforms, spam, etc.

3. Security Hardening

Security hardening is an informally known term right now and, as such, we first provide a definition for it. We also propose a taxonomy of security hardening methods that refer to area to which the solution is applied. We established our taxonomy by studying the solutions of software security problems in the literature. Even though our reading included a significant biais towards C [3, 9, 5, 10], we believe that our taxonomy is language independent. We also investigated the security engineering of applications at different levels, including specification and design issues [4, 5].

We define software security hardening as any process, methodology, product or combination thereof that is used to directly increase the security of existing software. In this context, the following constitutes the detailed classification of security hardening methods:
Code-Level Hardening are changes in the source code in a way that prevents vulnerabilities without altering the design. Some vulnerabilities are a direct result of the programming activities. Code level hardening constitutes of removing these vulnerabilities in a systematic way.
Software Process Hardening is replacement of the development tools and compilers, the use of stronger implementation of libraries and the use of code weaving tools in a way that does not change the original code yet results in increased security.
Design-Level Hardening consists of the reengineering of the application in order to integrate security features that were absent or insufficient. Some security vulnerabilities cannot be resolved by a simple change in the code or by a better environment, but are due to a fundamentally flawed design. This category of hardening practices target more high-level security such as access control, authentication and secure communication. In this context, best practices and security design patterns [4], can be redirected from their original intent and used to guide the redesign effort.
Operating Environment Hardening stands for improvements to the security of the execution context (network, operating systems, libraries, etc.) that is relied upon by the software. Those changes make exploitation of vulnerabilities typically harder, although they do not remedy them. [1, 10].

4. Hardening for High Level Security

This section illustrates our proposition and methods to harden high level security into applications. At that level, deploying such type of security is mainly categorized as design-level hardening. For this reason, some methods of security design and application re-engineering will be needed to achieve our goal. Our approach for hardening of high level security uses security ontology to enable the automated processing of security related information. Ontology is the specification of a conceptualization of a knowledge domain. Security ontology provides definition of security concepts and relations between them. These are required as many different definitions are used in security literature, so a clarification of the relations between the concepts help to get a better understanding of the overall security concept. The security ontology used in this paper is part of a complete one presented in [8]. We chose this ontology because it describes a generic model of security applied in all the domains. As such, these definitions can be modified to fit in other specific application domains. Figure 1 illustrates those concepts and shows the relationships among them.

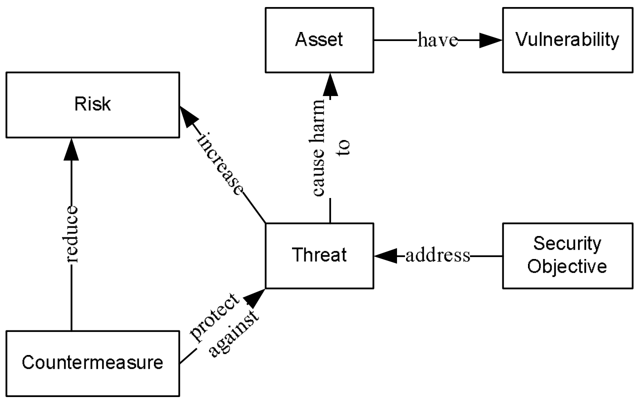

Figure 1. Security Ontology Architecture

The following is a brief description of this security ontology's components and their relationships:
Assets are information or resources which have value to an organization or person. Applications, systems and networks are counted as assets. The weak assets are those that have vulnerabilities.

Vulnerabilities are flaws or weaknesses in an asset. Attackers exploit vulnerabilities to break the security of an asset.
Risks are the probabilities that an attack to an asset succeeds. Threats increase the risk for security breach, while countermeasures reduce it.
Security Objectives are statement of intents to counter and address threats and satisfy the identified security needs. The state of security is achieved when the protection against threats is guaranteed.
Threats are exploitable vulnerabilities. Threats cause harm to assets and increase the risk of security breach.
Countermeasures are mitigation techniques performed in order to protect an asset against threats and attacks. Countermeasures reduce the risk of security breach.

Expressing the security concept in such an ontological way permit us to better identify and order the steps needed to harden security into applications. This ontology can be converted into a process, where each entity dependent on another becomes an analysis step dependent on the results of the previous related step(s). Although security hardening focuses on finding the best countermeasure to a particular threat, we have no choice to perform some prerequisite tasks in order to achieve a complete hardening process. In this context, we present in the following subsections the steps needed to harden security into applications.

4.1. Identifying Threats and Calculating Risks

Identifying threats is an important task in security hardening since we need to determine which treats require mitigation and how to mitigate them, preferably by applying a structured and formal mechanism or process. As such, the following is a brief description of the three main steps needed to identify and evaluate the risk of a threat:
Application Decomposition is dividing the application into its key components to identify the trust boundaries between them. This decomposition help to minimize the number of threats that need mitigation by excluding those that are outside the scope and beyond the control of the application.
Threat Identification is mainly categorizing it with respect to the six know categories presented in [5]: Spoofing identity, tampering with data, repudiation, information disclosure, denial of service and elevation of privilege.
Risk Evaluation is needed to determine the priority of threats to be mitigated.

4.2. Countermeasures

Once the previous steps are done and the threat is well identified and categorized, it is possible to determine the appropriate technique(s) to mitigate it. In the literature, it is possible to find mapping between categories of threats and known counter-measures addressing them. Choosing the best techniques will be mostly based on the state of the art of weaknesses and mitigation methods as well as security patterns. In [5], the authors provide a list of mitigation techniques for each category of threats of their classification. For example, against the threat of Spoofing Identity, they recommend to use appropriate authentication and to protect secret data; against Information Disclosure, they recommend to use authorization, encryption and to protect secrets, etc. Regarding the deployment of these techniques into applications and systems, security patterns are a useful to choose the best techniques available, and guide their implementation.

5. Hardening of Low Level Security

As a starting point, we discuss in this section the major safety vulnerabilities of C programming that are introduced in the source code during the implementation. In each of the following subsections, we describe briefly each vulnerability as well as the hardening techniques used to remedy them.

5.1. Hardening for Buffer Overflow Vulnerabilities

Buffer Overflows (BoF) exploit common programming errors that arise mostly from weak or non-existent bounds checking of input being stored in memory buffers. Buffers on both the stack and the heap can be corrupted [5]. Many APIs and tools have been deployed to solve the problem of Buffer Overflow or to make its exploitation harder [2, 6]. More methods for secure coding can be found in [5, 3]. In this context, Table 1 summarizes the security hardening solutions for buffer overflows.

Hardening Level	**Product/Method**
Code	Bound-checking, memory manipulation functions with length parameter, null-termination, ensuring proper loop bounds, format string specification, input validation
Software Process	Compile with canary words, inject bound-checking aspects
Design	Input validation, input sanitization
Operating Environment	Disable stack execution, use libsafe, enable stack randomization

Table 1. Hardening for Buffer Overflows

5.2. Hardening for Integer Vulnerabilities

Integer security issues are caused by converting between signed and unsigned, signedness errors, truncation errors and overflow and underflow [9]. Those vulnerabilities can be solved using sound coding practices and special features in some compilers (i.e. replace integer operations with safer calls)[9]. The security hardening solutions for such problems are summarized in Table 2.

Hardening Level	Product/Method
Code	Use of functions detecting integer overflow/underflow, migration to unsigned integers, ensuring integer data size in assignments/casts
Software Process	Compiler option to convert arithmetic operation to error condition-detecting functions

Table 2. Hardening for Integer Vulnerabilities

5.3. Hardening for Memory Management Vulnerabilities

The C programmer is in charge of pointer management, buffer dimensions, allocation and deallocation of dynamic memory space, which may cause memory corruption, unauthorized access to memory space, buffer overflow, etc [9]. Security hardening solutions against such problems are summarized in Table 3.

Hardening Level	Product/Method
Code	`NULL` assignment on freeing and initialization, error handling on allocation, pointer initialization, avoid null dereferencing
Software Process	Using aspects to inject error handling and assignments, compiler option to force detection of multiple-free errors
Operating Environment	Use a hardened memory manager (e.g. dmalloc, phkmalloc)

Table 3. Hardening for Memory Management Vulnerabilities

5.4. Hardening for File Management Vulnerabilities

File management errors can lead to the many security vulnerabilities such as data disclosure, data corruption, code injection and denial of service. Unsafe temporary file and improper file creation access control flags are two major sources of vulnerabilities in file management [10]. In some cases, we can redesign the application to use inter-process communication instead of temporary files. The security hardening solutions for such problems are summarized in Table 4.

Hardening Level	Product/Method
Code	Use proper temporary file functions, default use of restrictive file permissions, setting a restrictive file creation mask, use of ISO/IEC TR 24731 functions
Software Process	Set a wrapper program changing file creation mask
Design	Redesign to avoid temporary files
Operating Environment	Restricting access rights to relevant directories

Table 4. Hardening for File Management Vulnerabilities

6. Conclusion

We defined in this paper the concept of software security hardening and a classification for hardening methods. Our proposition will guide the developers and maintainers to deploy and harden security features and remedy present vulnerabilities into existing open source software. Moreover, we summarized the common vulnerabilities found in the C language and their solutions using our framework, demonstrating its usability. This paper constitutes the first task of a complete security hardening framework.

References

[1] Bastille linux, 2006. `http://www.bastille-linux.org/`.

[2] S. Bhatkar, D. DuVarne, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[3] M. Bishop. How Attackers Break Programs, and How to Write More Secure Programs. `http://nob.cs.ucdavis.edu/bishop/secprog/sans2002/index.html`.

[4] B. Blakley and C. Heath. Security design patterns. Technical Report G031, Open Group, 2004.

[5] M. Howard and D. E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.

[6] J. McCormick. Openbsd declares war on buffer overruns. *TechRepublic*, 2003. `http://techrepublic.com.com/5100-1035_11-5034831.html`.

[7] OSTG. Sourceforge.net: Software map. `http://sourceforge.net/softwaremap/trove_list.php`.

[8] M. Schumacher. *Security Engineering with Patterns*. Springer, 2003.

[9] R. Seacord. *Secure Coding in C and C++*. SEI Series. Addison-Wesley, 2005.

[10] D. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010*. 2003. `http://www.dwheeler.com/secure-programs/`.