소프트웨어 공격에 대한 보안성 분석

김정태
목원대학교

Analyses of Security for Software Attack

Jung-Tae Kim
Mokwon University
E-mail : jtkim3050@mokwon.ac.kr

요 약

Software security is about making software behave correctly in the presence of a malicious attack, even though software failures usually happen spontaneously in the real world. Standard software testing literature is concerned only with what happens when software fails, regardless of intent. The difference between software safety and software security is therefor the presence of an intelligent adversary bent on breaking the system. Software security for attacking the system is presented in this paper

Ⅰ. Introduction

System Safety Engineering is the application of scientific and Systems Engineering methodology and management techniques to minimize risk in a product. Software safety is the extension of that discipline into software. Significant differences exist between the traditional safety approach to hardware control and failure modes and safety's new approach to software control and failure modes. While the management shell existed, there were few analysis techniques which were applicable five years ago. Today the safety management shell is being successfully integrated into the software realm and several analysis techniques have nearly been perfected, showing independently repeatable results. The current iteration of improvement involves determining the type and timing of specific data from Software and quality engineers.

II. Software Security Knowledge Objects

Figure 1 shows the seven distinct knowledge catalogs (the boxes) that we divide into three knowledge categories. The category prescriptive knowledge includes three knowledge catalogs: principles, guidelines, and rules. These sets span a continuum of abstraction from high-level architectural principles at the philosophical level (for example, the principle of least privilege1) to very specific and tactical code-level rules (for example, "avoid the use of the library function gets() in C"). Guidelines fall somewhere in the middle of this continuum (for example, "make all Java objects and classes final(), unless there's a good reason not to"2). As a whole, the prescriptive knowledge category offers advice for what to do and what to avoid when building secure software. The diagnostic knowledge category includes three knowledge catalogs: attack patterns, exploits, and vulnerabilities. Rather than prescriptive statements of practice, diagnostic knowledge helps practitioners

(including those working in operations) recognize and deal with common problems that lead to security attacks. Vulnerability knowledge includes descriptions of software vulnerabilities experienced and reported in real systems (often with a bias toward operations). Exploits describe how instances of vulnerabilities are leveraged into particular security compromise for particular systems. Finally, attack patterns describe common sets of exploits in a more abstract form that's applicable across multiple systems. Such diagnostic knowledge is particularly useful in the hands of a security analyst, although its value as a resource to be applied during development is considerable (consider, for example, the utility of attack patterns to abuse case development.)

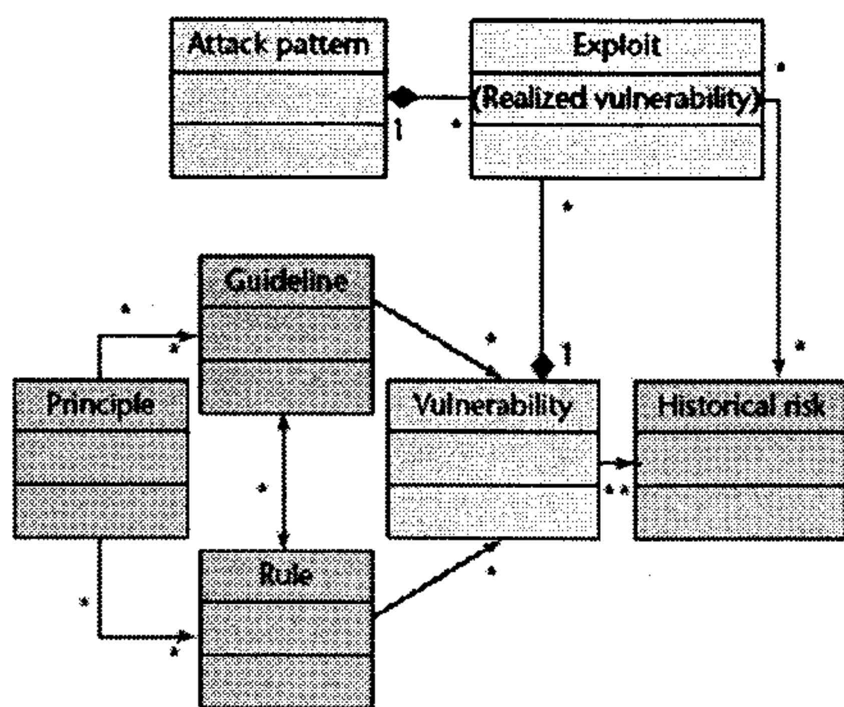

Fig. 1 Software security knowledge objects and a basic interrelating architecture.

III. Security Mechanism

Software Safety Engineering has become a required business endeavour. It has been executed on aircraft, missiles, warheads, medical devices, trains, and should soon be making it's way into communications systems and even grocery stores. As fast as an analysis techniques were derived, applied, and corrected, technology took still another step forward, perhaps even on the same development project! Meanwhile, we developed management plans which allowed us to integrate several design, development, analysis, and test techniques over an entire product life cycle. This has provided a solid baseline from which to extend and continuously improve. Software safety attempts to guide design through requirements and trade studies to infuse safety and defuse hazards from the start. The identifying and assessing of risk is done so that we can allocate resources against safety-critical code and determine remaining risk. An important service which helps lower risk is to bring the specific hazards to the attention of the appropriate designer or coder and to brief management and the customers. Once Safety identifies the initial hazards, requirements are levied against those requiring fixing, and in what priority. The key reason is to eliminate or control the hazard. Later analyses will statically ensure the hazard has indeed been controlled and no other hazards were inadvertently introduced by implementation. Dynamic testing can prove that a hazard actually was designed out as the requirement states. Without requirements, however, there probably will not be specific code addressing the hazard nor tests to illustrate the removal of the hazard. Unless specific contracting language includes it, separate safety testing will not occur. Imagine an Application Specific Integrated Circuit(ASIC) which is well into manufacturing layup when a safety hazard is discovered, which the customer demands be fixed. There will be perhaps hundreds of thousands in engineering, hours, manufacturing, test plans, and documentation reworked while the hazard is removed or mitigated. A special test point may have to be drawn out of the circuit to allow testing that device on-line

or to insert a fault in functional qualification tests. The identity of each hazard must be known and the

requirement against it must be testable (have some external lead perhaps). These two points need to be front-loaded (designed-in) or it will cost the company money to step back and fix it. Concurrent engineering will either live or die (economically) by the ability to find hazards early.

IV. Software Development Issues

Landwehr's genesis provided the basis for describing the way each vulnerability entered the system. In general it is the type of security flaw, which is present:

- Validation errors occur when a program fails to check that the parameters supplied or returned to it conform to its assumptions about them, or when these checks are misplaced.
- Domain errors occur when the intended boundaries between protection environments are porous including implicit sharing of privileged/confidential data or when then the lower level representation of an abstract object, supposed to be hidden in the current domain, is in fact exposed.
- Serialization flaws permit asynchronous behavior of different system components to be exploited. Many time of check-to-time-of-use (TOCTTOU) flaws fall in this category. Aliasing flaws arise when two names for the same object can cause its contents to change unexpectedly and consequently, invalidate checks already applied to it. Serialization and aliasing flaws are combined into one category.
- An identification/authentication flaw permits operations to be invoked without sufficiently checking the identity and the authority of the invoking entity.
- Boundary condition flaws occur due to omission of checks to assure that constraints (table size, file allocation, or other resource consumption) are not exceeded.
- Trojan horse refers to a program that masquerades as a useful service but exploits rights of the program's user rights not possessed by the author of the Trojan horse - in a way the user does not intend.
- Covert Channel is defined as a path to transfer information in a way not intended by the system's designers.
- Other Exploitable logic errors include all errors that do not fall in any of the above categories. We have simplified this categorization by not distinguishing between intentional and inadvertent as well as malicious or non-malicious flaws in Landwehr's taxonomy. From a testing perspective, it is essential to test systems adequately to discover such security flaws, and the programmer's intent is not important in this context.
- Exploitable logic errors occur due to use of incorrect logic during implementation. Our simplified genesis dimension is then: validation errors, domain errors, serialization or aliasing errors, errors due to inadequate identification or authentication, boundary and condition errors, Trojan horse, covert channel and exploitable logic

V. Security Features

Software security isn't security software. All the magic crypto fairy dust in the world won't make your code secure, but it's also true that you can drop the ball when it comes to essential security features. Let's say you decide to use the Secure Sockets Layer (SSL) to protect traffic across the network, but you really screw things up. Unfortunately, this happens all the time. When we chunk together security features, we're concerned with topics like authentication, access control, confidentiality, cryptography, privilege management, and all that other stuff on the CISSP exam. This stuff is hard to get right. Software security can and should borrow from other disciplines in computer science and software engineering when developing and evolving best practices. Of particular relevance are

- security requirements engineering,
- design for security, software architecture, and architectural analysis,
- security analysis, security testing, and use of the Common Criteria,
- guiding principles for software security and case studies in design and analysis,
- auditing software for implementation risks, architectural risks, automated tools, and technology developments (code scanning, information flow and so on), and
- common implementation risks (buffer overflows, race conditions, randomness, authentication systems, access control, applied cryptography, and trust management).

VI. Important Security Features from Protection Profile

A Protection Profile is an implementation independent set of security requirements for a category of targets that meet specific consumer needs. When Target of Evaluation (TOE) is used D make up a larger system environment. the boundaty protection must provide the appropriate security mechanisms, cryptographic strengths and assurances to ensure adequate protection for the security and integrity of this TOE. This Protection Profile could be viewed as a roadmap that finds what features a Trusted Operating System should have. We study the protection profile, and find six important security features of these security requirements for a Trusted Operating System. They are (I) ldenrrficdiun ond Aurhmricnlion. (2) Discrerionarv Access Conrrol. (3) Mandurvrv Access Control. (4) Mandarory lnregriry Control. (5) Cryptographic services, (6) Audir services.

VII. Conclusion

We give an approach for teaching appropriate security-aware concepts in a software curriculum and map the skills and concepts to specific courses. Software vulnerability is second only to identity theft as the main security problem of the modern Internet. We propose an approach to reversing the trend that is inexpensive and consistent with existing and known successful programming practice.

References

[1] David Aucsmith, "Tamper Resistant Software: AnImplementation", Proceedings of the First International Workshop on Information Hiding, Pages: 317-33, 1996, LNCS 1174

[2] Toshio Ogiso ,Yusuke Sakabe,Masakazu Soshi,and Atsuko Miyaji, "Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis", WISA 2002, Cheju Island, Korea, August 28-30, 2002

[3] G. Hoglund and G. McGraw, Exploiting Software: How to Break Code, Addison-Wesley, 2004.