Ferret: A Host Vulnerability Checking Tool*

Anil Sharma†, Jason R. Martin‡, Nitin Anand†, Michel Cukier†, and William H. Sanders‡

Center for Reliability Engineering†
Department of Mechanical Engineering
University of Maryland at College Park
College Park, Maryland 20742
{asharm, nitina, mcukier}@eng.umd.edu

Coordinated Science Laboratory‡
Department of Electrical & Computer Engineering,
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
{jrmartn1, whs}@crhc.uiuc.edu

Abstract

Evaluation of computing system security requires knowledge of the vulnerabilities present in the system and of potential attacks against the system. Vulnerabilities can be classified based on their location as application vulnerabilities, network vulnerabilities, or host vulnerabilities. This paper describes Ferret, a new software tool for checking host vulnerabilities. Ferret helps system administrators by quickly finding vulnerabilities that are present on a host. It is designed and implemented in a modular way: a different plug-in module is used for each vulnerability checked, and each possible output format is specified by a plug-in module. As a result, Ferret is extensible, and can easily be kept up-to-date through addition of checks for new vulnerabilities as they are discovered; the modular approach also makes it easy to provide specific configurations of Ferret tailored to specific operating systems or use environments. Ferret is a freely available open-source software implemented in Perl.

Keywords: Security auditing tool, host vulnerabilities, security evaluation.

1. Introduction

Historically, approaches to security validation have focused on specification of procedures that should be followed during the design of a system (e.g., the Security Evaluation Criteria [3, 7]). When quantitative methods have been used, they have typically either been based on formal methods (e.g., [10]), aiming to prove that certain security properties hold given a specified set of assumptions, or been quite informal, using a team of experts (often called a "red team," e.g. [11]) to try to compromise a system.

An alternative approach, which has received much less attention from the security community, has been to try to quantify, probabilistically, the behavior of an attacker and his/her impact on the ability of a system to provide certain security-related properties. For example, Gong et al. [5] presented a general 9-state model of an intrusion-tolerant system. In a different approach, Jha and Wing [8] proposed that state-level modeling, formal logic, and a Bayesian analysis be used together to quantify the survivability of a system. Ortalo et al. [15] proposed formal description of known vulnerabilities present in a system using a "privilege graph." Finally, [17, 6] used stochastic activity networks to evaluate the intrusion tolerance of several realistic systems. All these approaches made important contributions to probabilistic evaluation of system security, but none proposed a complete framework based on experimental data. [16] presents a probabilistic framework for probabilistically validating security. This probabilistic framework has two components: 1) a model of an attacker, the system, and the workload demanded of a system, and 2) a set of measurements that provide estimates of the values of model parameters. The purpose of this paper is to describe a new tool for experimentally quantifying host vulnerabilities, which can 1) provide input parameter values for models constructed according to the probabilistic security validation framework described in [16], and 2) provide useful information to system administrators who wish to eliminate vulnerabilities.

Precise definitions of several terms will help to clarify the challenge in building such a tool. In the following, we adopt the terminology of [12, 13]. In particular, an *attack* is a malicious act that attempts to exploit a weakness in the system. Such a weakness is called a (security) *vulnerability*, which is an accidental fault or a malicious or non-malicious intentional fault. An *intrusion* results from an attack that has been (at least partially) successful. An attack is thus an

* This research has been supported by DARPA contract F30602-00-C-0172, NSF ITR contract 0086096, and NSF CAREER award 0237493.

intrusion attempt. An intrusion can thus be seen as the exploitation of a vulnerability. Vulnerabilities are usually classified into *application vulnerabilities*, *network vulnerabilities*, and *host vulnerabilities*.

The security community had a great interest in host vulnerabilities about ten years ago. Since then, the focus has shifted towards network vulnerabilities because of the assumption that once an outsider has found a way to penetrate the network, it is very difficult to keep him or her from getting administrative privilege on hosts [19]. This argument may make sense if one is focusing on the problem of attacks from outside, but it becomes less convincing when one is trying to evaluate the security of a computer network with respect to the possibility of both internal and external attack. In that case, host as well as network and application vulnerabilities need to be considered.

Host vulnerabilities are considered in a broad sense in this paper. Some might argue that some of the vulnerabilities are in fact configuration errors or that some of the vulnerabilities are features provided to the user. Despite such arguments, we simply use the term "vulnerability" for anything identified as potentially exploitable by attackers.

Several tools have been developed to check host vulnerabilities (e.g., COPS [4] and Tiger [18]). Having been developed about ten years ago, COPS and Tiger have several limitations. For example, they check for some vulnerabilities that are no longer relevant, and offer no simple way to modify the list of checked vulnerabilities. Furthermore, they do not check certain important current vulnerabilities. Moreover, their implementation is too customized to specific Unix OSs and versions. For example, they need to be modified in order to run on current versions of Linux. Finally, they were not designed using a modular approach (making updates to checked vulnerabilities difficult). For all these reasons, we designed a new tool, called *Ferret*, for checking host vulnerabilities. Nessus [14], a recently developed network vulnerability-checking tool, motivated several of the design principles we followed in constructing Ferret. Nessus is a free open-source tool, is based on plug-in modules (each plug-in module checks a specific vulnerability), and is updated daily by the Nessus community. The security community would clearly benefit from having a tool for checking host vulnerabilities that is similar to Nessus.

We have thus designed Ferret in a modular way: a different plug-in module is used for each vulnerability checked, and the format of the output is also specified by different plug-in modules. As a result, Ferret is extensible, and can easily be kept up-to-date through addition of checks for new vulnerabilities as they are discovered; the modular approach also makes it easy to provide specific configurations of Ferret tailored to specific operating systems or use environments. Ferret is a freely available open source software implemented in Perl [1]. This will facilitate the rapid development of new vulnerability-checking plug-in modules when new vulnerabilities are found. So that Ferret can run on various platforms, its management component is designed to be platform-independent, and as many of the vulnerability-checking plug-in modules as possible are also platform-independent. In some cases, the vulnerability checked is specific to one platform, and thus the associated plug-in module needs to be platform-dependent. It is easy to group vulnerability-checking plug-in modules using keywords designed in the Ferret management component. The output plug-in modules also provide flexibility by creating reports, including raw text files.

The remainder of the paper is organized as follows. The detailed design, configuration, and implementation details of Ferret are described in Section 2. Section 3 contains some preliminary results of a data collection conducted using Ferret. Finally, Section 4 presents several conclusions.

2. Ferret Detailed Design

Before describing the architecture of Ferret, we explain our motivation for choosing an interpreted language, Perl, for implementing Ferret. Since Ferret has been designed to process text files, the use of Perl allows us to take advantage of its well-known text-processing capabilities. Moreover, Perl is available on many popular platforms. The interpreted nature of Perl also promotes simplicity of use without complicated compilation procedures. Perl has also been shown to be efficient despite being an interpreted language. That is important because it makes it possible to scan large systems within a reasonable time. Lastly, Perl is a very powerful and expressive language (e.g., its high-level data structures, like hash tables, are very helpful).

2.1. Architecture Overview

Ferret's structure includes three major areas: the management (Ferret core), the scanning agent (set of vulnerability-checking plug-ins, with individual plug-ins looking for specific vulnerabilities), and the collecting agent (set of output plug-ins). The Ferret management core runs the vulnerability-checking plug-ins, collects the results, and uses the selected output plug-in to provide a report in the desired format. The structure of Ferret is illustrated in Figure 1.

The communication interface between the Ferret core and the plug-ins consists of command-line arguments. Standard command-line options are used to pass the options between the core and the plug-ins. This approach was chosen over the use of Perl packages/modules in order to allow

1 Ferret is available at http://ferret.crhc.uiuc.edu

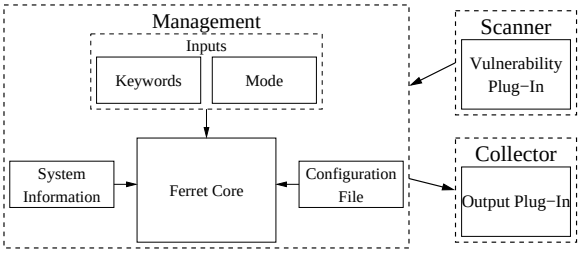


Figure 1. Ferret Architecture Overview

each plug-in to be run as a stand-alone program, should a system administrator prefer that approach to using the Ferret core. Output results can be saved as a raw data file and piped through one of the output plug-ins at a later time to interpret the results off-line. The location of the vulnerability-checking plug-ins and the output plug-ins can be passed as a command-line option. By default, to obtain the location information, the Ferret core looks in the configuration file.

The inputs include keywords (used to select subsets of vulnerability-checking plug-ins to run), the level of verbosity (used to help system administrators in making the implementation more transparent), and the modes used to run the Ferret core. The selection of the mode can be passed as a command-line option. The choice of the mode impacts the information provided in the output file. Two modes are available in the current version of Ferret. The first mode has been designed to focus specifically on which vulnerability was found on the host. The second mode has been designed to provide some more detailed information on the vulnerabilities found, such as how the vulnerability can be exploited. This mode has been customized so that privilege graphs [1] can be built from the output results to allow later assessment of the security of the computing system.

In the following sections, we detail the main components of Ferret: the core, the vulnerability-checking plug-ins, and the output plug-ins.

2.2. Ferret Core

The Ferret core was developed to be platform-independent and simple. This section details the different steps executed by the Ferret core.

1. It interprets the command line options and reads the configuration file.
2. It determines information about the system Ferret is running on, such as the operating system name and version. This information is helpful in later stages in filtering out the plug-ins that should remain active for a particular host. For example, a plug-in developed specifically for the Linux system should not be executed on a Solaris machine.
3. It determines what plug-ins are available in the system from the directory defined in the configuration file (or on the command line).
4. It queries each plug-in for information about what vulnerability it checks, what level of privilege this vulnerability could gain, on which operating system this vulnerability should be checked for, and what keywords are applicable to this plug-in. This information is stored in a hash table for easy reference later.
5. It builds a hash table of the keywords, based on individual keywords, in order to simplify the running of the plug-ins.
6. It uses the keywords to determine which plug-ins to run. If no keyword is provided, then a default keyword of *all* is used to run all the plug-ins.
7. It runs the chosen plug-ins in sequence.
8. It divides plug-ins, based on the results obtained, into two categories: those that have found vulnerabilities, and those that have not.
9. It pipes out the results, based on the mode selected by the user, through the corresponding output plug-ins in the desired format.

2.3. Vulnerability-Checking Plug-ins

2.3.1. Options for Running the Vulnerability-Checking Plug-ins The goal of each vulnerability-checking plug-in is to scan the system for a particular vulnerability. Using the command line options, the Ferret core makes the individual plug-ins perform diverse functions. For example, when Ferret is executed with the –info option, each plug-in returns information including its version number, a list of keywords associated with this plug-in, the list of operating systems on which the plug-in can be run, and a short description of the vulnerability the plug-in checks.

Vulnerability-checking plug-ins can also be written to interface with some other tools. Such plug-ins then pass the obtained output results to the Ferret core. For example, a plug-in has been written to instrument the commonly used password-cracker tool, John the Ripper [9], to run for a specified time to identify passwords that are be easily guessable.

Finally, depending on their needs, system administrators can enable or disable the execution of plug-ins, making it possible to execute only a specific subset of plug-ins. For example, plug-ins checking SUID files may take a lot of time because they need to search through the filesystem.

2.3.2. Groups of Vulnerability-Checking Plug-ins Based on the type of vulnerability a plug-in checks for, plug-ins have been divided into various groups. Vulnerabilities belonging to a particular group have similar

characteristics. Currently, about 80 plug-ins have been implemented.

The first group of plug-ins checks whether or not certain critical system files or directories (e.g., /bin, /.cshrc, /dev, /etc/group, /etc/passwd, /etc, /.login, /.profile, /.rhosts, /usr/etc) are owned by root. The second group of plug-ins checks if the permissions of critical system directories (e.g., /, /bin, /usr/adm, /etc, /dev) are world-writable. The third group of plug-ins checks for the paths and filenames inside the root start-up files (e.g., /.login, /.cshrc, /.profile) for world-writability. The fourth group of plug-ins checks the umask settings in login initialization files (e.g., /.login, /cshrc, /.profile, /etc/profile). The fifth group of plug-ins focuses on configuration issues for certain files in each user's home directory. More specifically, these plug-ins check whether permissions on certain important files (e.g., .login, .logout, .profile, .rhosts, .tcshrc, .xinitrc) of any user's home directory are set to group- or world-writable. In Unix systems, the SUID bit allows a file to be run as if by the user who owns the file, even if it is actually being run by someone else. The sixth group of plug-ins is designed to search for all the files on the target system that have the SUID bit enabled. The seventh group of plug-ins focuses on the different fields of the password file. The eighth group of plug-ins focuses on system file permissions. Some of the plug-ins developed in this family target the /etc/exports file, and others target the /etc/fstab file. The ninth group of plug-ins checks the .rhosts file to see if any unreasonable permission is given to remote users and machines.

2.4. Output Plug-ins

As mentioned earlier, the Ferret core, the vulnerability-checking plug-ins, and the output plug-ins can be run in different modes. For each of the modes, the Ferret core generates a raw format output file using the output from different vulnerability-checking plug-ins. Each mode can have a number of different formats in which to present the final output results generated by that mode. For example, a user could generate a report in XML format by defining its attributes in an XML output plug-in. As mentioned earlier, two modes have been implemented. The first mode has been designed to focus specifically on information on vulnerabilities. The second mode has been designed to provide some more detailed information on the vulnerabilities found and their possible exploitations for the purpose of quantifying the security.

2.4.1. Output Plug-in with First Mode The raw format output file created by the Ferret core for this mode consists of four fields. The first field contains the name of the plug-in. The second field contains a short description of what the plug-in does. The third field indicates the result of the scan by the plug-in. It specifies whether a vulnerability was found. The last field provides some more information relevant to the vulnerability found by the plug-in. For example, if a plug-in finds that the home directory in a password file is group-writable, then the output field will list the exact home directory that is group-writable.

This mode is useful to the system administrator for determining various vulnerabilities present in the system, including those in users' home directories. Based on the obtained results, an organization security policy can be framed out, and users' negligence can be effectively checked.

2.4.2. Output Plug-in with Second Mode The raw format output file created by the Ferret core for this output plug-in consists of the list of all the users (along with their groups) present in the home directory of the host on which Ferret is running, plus four additional fields. The first field contains the name of the plug-in. The second field contains information about any possible change in the privilege associated with the vulnerability. The third field provides the name of the file or directory that is vulnerable. The last field gives a brief description of the vulnerability.

Through the use of a privilege graph, the information provided by this output plug-in on the vulnerability and its exploitation through the possible change of privilege can be used to evaluate the security of a computing system. The way we do so is based on work by Ortalo et al. [15], who presented a method for evaluating security by building a privilege graph, and by Dacier and Deswarte [1], who demonstrated that host vulnerabilities can be represented in a privilege graph. In such a graph, each node represents a set of privileges assigned to a user or set of users (e.g., a Unix group). Any arc that connects two nodes is representative of a vulnerability, which allows a user (starting node) to achieve another user's privilege (ending node).

The output file provided by Ferret contains the different elements for building a privilege graph. The list of all users, along with their groups contained in the output file, defines the set of nodes. The set also includes two special nodes that represent a user with no privileges (an outsider) and a user who has administrative privileges. A transition between two nodes is associated with each vulnerability. The obtained output file contains the list of vulnerabilities found on a host, and thus contains the list of transitions between two nodes in the privilege graph. The output plug-in with the second mode thus contains the textual description of a privilege graph.

3. Experimental Results

In order to illustrate the performance and usefulness of Ferret, we describe in this section some results obtained from a short data collection. Ferret has been installed on six machines at the University of Illinois at Urbana-Champaign. Data have been collected three days a week

for one and a half months. The chosen machines are different Unix operating systems and versions, including three server-class machines and three workstations. Table 1 summarizes both the average number of types of vulnerabilities and the average number of vulnerabilities of each type, for the period of data collection.

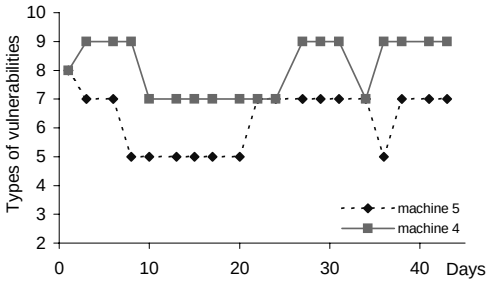
3.1. Discussion of Scan Results

The large number of vulnerabilities found on the HP-UX server is worth noticing. One explanation could be that this machine is relatively old (i.e., about 3.5 years). Another reason could be the large number of users it is serving (i.e., about 335 users). During the data collection, most of the vulnerabilities detected by Ferret arose from non-root ownership of critical files and directories (first group), configuration issues in certain files of users' home directories (fifth group), and vulnerabilities related to the password file (seventh group).

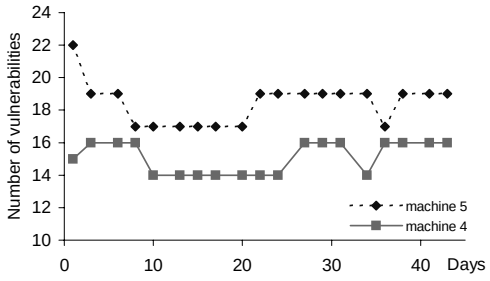
Some system-critical files and directories (like /bin, /dev, /etc) were not owned by root on machines 1 and 6. Plug-ins focusing on configuration issues in home directory files accounted for the highest number of vulnerabilities. The vulnerabilities found by plug-ins focusing on the non-ownership by root of critical files/directories and on vulnerabilities in the password file are very serious in nature and demand more alertness on the part of the system administrators. On the other hand, the significant number of vulnerabilities found related to configuration issues in home directory files shows negligence on the part of users.

Curves have been plotted for machines 4 and 5 to show the variation in both types of vulnerabilities, and the numbers of each vulnerability type with the number of days are shown in Figure 2. The period of data collection is too small to obtain any significant trend, but still, certain changes can be observed. For example, for both hosts there were five to ten types of vulnerabilities present. While some of the serious vulnerabilities (like /etc file not owned by root) were present all the time, the fluctuation that did occur was mainly due to changes in the permission settings of certain files in user home directories. On the other machines, the vulnerabilities showed a static behavior. For example, for machine number 6, the number of vulnerabilities remained static throughout the period of observation.

Most of the vulnerabilities found are easily correctable, and a system administrator with a tool such as Ferret could use it to check the machines quickly (with an average time of around 120 seconds) and efficiently on a regular basis.



(a) Variation in Type of Vulnerabilities Over Time



(b) Total Number of Vulnerabilities

Figure 2. Results

4. Conclusion

Ferret has been developed with the idea of providing the security community with a useful and efficient tool for checking for host vulnerabilities. It is a freely available open source tool developed in Perl. It consists of a set of plug-in modules. Each host vulnerability is checked by one plug-in module. Moreover, the output provided by Ferret is also specified through the use of plug-in modules. Currently, about 80 plug-in modules have been developed. They mainly target Linux, Solaris, and HP-UX vulnerabilities. We expect the security community to participate in developing new plug-in modules for multiple operating systems. System administrators could then select subsets of plug-in modules to check for vulnerabilities customized to their computer networks. Moreover, even though the vulnerabilities currently addressed focus on Unix platforms, the development of Ferret in Perl facilitates the introduction of plug-in modules targeting vulnerabilities in Windows.

Acknowledgments

The authors would like to thank Michael Chan for fruitful discussions on Ferret and for facilitating the use of Ferret in a production environment. We would like to thank Tod Courtney for running early versions of Ferret and providing useful feedback. We would also like to thank the other

Machine	OS	OS Version	Machine Function	Vulnerability Types	Number of Vulnerabilities	Execution Time (sec)
1	Linux	2.4.9-31 (RedHat 7.2 with updates)	Workstation	22	102	146
2	Sun Solaris	5.8	Workstation	2	4	29
3	Sun Solaris	5.8	Server (NIS, NFS, SunRPC services)	12	14	234
4	Sun Solaris	5.8	Server (NIS, NFS)	8	19	180
5	Sun Solaris	5.7	Workstation	6	15	25
6	HP-UX	B.10.20	Server (NIS, NFS, DNS, DHCP, Mail)	35	303	111

Table 1. Test Machine Configuration

members of the ITUA team for their helpful comments. We are grateful to Jenny Applequist for her editorial assistance.

References

[1] M. Dacier and Y. Deswarte, Privilege Graph: an Extension to the Typed Access Matrix Model, in Proc. Third European Symposium Research in Computer Security (ESORICS94), pp. 317-334, 1994.

[2] M. Dacier, Y. Deswarte, and M. Kaâniche, Quantitative Assessment of Operational Security: Models and Tools, LAAS Research Report 96493, May 1996.

[3] U.S Department of Defense Standard, Department of Defense Trusted Computer System Evaluation Criteria (Orange Book), DOD 5200.28-STD, Library No. S225, 7ll, Dec. 1985. http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html

[4] D. Farmer and E. H. Spafford, The COPS Security Checker System, in Proc. Summer Usenix Conference, Berkeley, CA, USA, pp. 165-170, 1990.

[5] F. Gong, K. Goseva-Popstojanova, F. Wang, R. Wang, K. Vaidyanathan, K. Trivedi, and B. Muthusamy, Characterizing Intrusion Tolerant Systems Using A State Transition Model, in Proc. DARPA Information Survivability Conference and Exposition II. DISCEX'01, 2001.

[6] V. Gupta, V. Lam, H. V. Ramasamy, W. H. Sanders, and S. Singh, Dependability and Performance Evaluation of Intrusion-Tolerant Server Architectures, in Dependable Computing: Proceedings of the First Latin-American Symposium (LADC 2003), São Paulo, Brazil, October 21-24, 2003, Lecture Notes in Computer Science vol. 2847 (Rogério de Lemos, Taisy Silva Weber, and João Batista Camargo Jr., eds), Berlin: Springer, 2003, pp. 81-101.

[7] ISO/IEC International Standards (IS) 15408-1:1999, 15408-2:1999, and 15408-3:1999, Common Criteria for Information Technology Security Evaluation: Part 1: Introduction and General Model, Part 2: Security Functional Requirements, and Part 3: Security Assurance Requirements, Version 2.1, August 1999 (CCIMB-99-031, CCIMB-99-032, and CCIMB-99-033).

[8] S. Jha and J. M. Wing, Survivability Analysis of Networked Systems, in Proc. of the 23rd International Conference on Software Engineering (ICSE 2001), p. 307-317, 2001.

[9] John the Ripper password cracker home page. http://www.openwall.com/john/

[10] C. Landwehr, Formal Models for Computer Security, Computer Surveys, vol. 13, no. 3, pp. 247-278, Sept. 1981.

[11] J. Lowry, An Initial Foray into Understanding Adversary Planning and Courses of Action, in Proc. DARPA Information Survivability Conference and Exposition II (DISCEX'01), pp. 123-133, 2001.

[12] C. Cachin, J. Camenisch, M. Dacier, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J.-C. Laprie, J.-C. Lebraud, D. Long, T. McCutcheon, J. Müller, et al., MAFTIA Reference Model and Use Cases, MAFTIA deliverable D1, 2000.

[13] MAFTIA Conceptual Model and Architecture, D. Powell and R. Stroud, Eds., MAFTIA deliverable D2, 2001.

[14] Nessus home page. http://www.nessus.org/

[15] R. Ortalo, Y. Deswarte, and M. Kaâniche, Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security, in IEEE Transactions on Software Engineering, vol. 25, no. 5, pp. 633-650, Sept.-Oct. 1999.

[16] W. H. Sanders, M. Cukier, F. Webber, P. Pal, and R. Watro, Probabilistic Validation of Intrusion Tolerance, in the Supplemental Volume of the 2002 International Conference on Dependable Systems and Networks (DSN-2002), Washington, DC, June 23-26, 2002, pp. B-78 to B-79.

[17] S. Singh, M. Cukier, and W. H. Sanders, Probabilistic Validation of an Intrusion-Tolerant Replication System, in Proc. 2003 International Conference on Dependable Systems and Networks (DSN-2003), San Francisco, CA, June 22-25, 2003, pp. 615-624.

[18] Tiger Analytical Research Assistant home page. http://www-arc.com/tara/index.shtml

[19] J. Viega and G. McGraw, *Building Secure Software*, Addison-Wesley, 2002.