

# DETECTING SOURCE CODE OF ATTACKS THAT INCREASE PRIVILEGE<sup>1</sup>

Robert K. Cunningham and Antonio Rieser  
Massachusetts Institute of Technology  
Lincoln Laboratory  
244 Wood Street  
Lexington, Massachusetts 02420-9185

Intrusion detection tools attempt to detect attacks that are underway or have already succeeded. Network probing and denial-of-service attacks often continue for some time before they are terminated, and this time window allows fast intrusion detection systems to alert before the attack is over. Attacks that increase privilege are much faster, sometimes only requiring a single command, system call or function. These attacks can be classified into two categories: those that provide access to an unauthorized user, or those that provide superuser access to a normal user.

Attacks that provide superuser access are often launched from the victim, allowing the attacker use the limited privileges available, and the defender to control and limit what comes onto the system. To effect such an attack, an intruder must either download or develop code, compile it, and use the compiled attack. Although not all steps need to be performed on the victim machine, sometimes source code will traverse a network that is being monitored by an intrusion detection system. We have performed an experiment to determine if attack code, developed either in C or in shell, could be accurately detected and differentiated from normal source code in a manner that does not merely detect specific attacks, but rather detects the underlying mechanisms required for an attack to succeed. If this proves successful, then we might have an approach that could be extended to detect binary code that grants a user increased privilege.

First, we acquired examples of freely available attack and normal code. For attack software, we downloaded all unix user-to-superuser attacks stored on rootshell.com from Jan 1 1998 to April 1 2000. This software was separated into shell (6 examples) and C code (20 examples). For normal shell software, we copied all start-up scripts that come with RedHat Linux 6.1 and are stored in `init.d` and in `rc[0123S].d`. For normal C software, we selected software that performs a range of tasks, including some operations that an attacker might perform. Normal C software included software that interacts with the filesystem (`fileutils-4.0`), software that interacts with a user (`shell-utils-2.0`), software that starts additional processes (`bash-2.04`, `gdb-4.18`), software that interacts with the network using various protocols (`sendmail-8.10.0`, `apache_1.3.12`), software that builds tables and graphs (`flex-2.54`) and software that interacts with the window manager, the user and provides an environment (`emacs 20.6`).

The normal and attack code was then examined, and potential features were identified. Regular expressions were written for about eighteen features. Shell and C features were similar in use, but different in implementation. Some of these were features for comments (attackers used `exploit`, `splloit`, or `vulnerability` in nearly every code example), system or shell utility calls (`link`, `unlink`, `exec`, `chmod`, `chown`, `chgrp`, in all their various forms), embedded low-level languages (shell code sometimes used C; C code sometimes used assembly), and commands that cleaned up after the attack (removing core files, touching installed trojans to make their times be correct).

---

<sup>1</sup> This work was sponsored by the Department of the Air Force under Air Force contract F19628-95-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Air Force.

The counts of these features were then gathered for every individual file listed above, and tests were performed using N-fold cross validation, with N set to the number of attack examples. Thus the classifier was trained using all attack features but one, and (N-1)/N of the normal examples, and tested using the remaining examples. The classifier is then re-trained for each of the remaining folds. To select features, a two-layer perceptron was trained using each individual feature, then that feature combined with all remaining features, etc., until the addition of new features resulted in no improvement in detection and false alarm rates. The two layer perceptrons each had an  $2i$  hidden units, where  $i$  is the number of inputs. Input prior probabilities of attack and normal examples were made equal to increase the number of correct detections, at the expense of increased false alarms.

For shell code, all attacks can be perfectly differentiated using just three features: the presence of an exploit comment, code to log into the current host, and code that changes the ownership of a file. This would be useful in detecting malicious users who download their code from attack repositories such as rootshell.com. Attackers do not need to comment their source, and comments will not be present in compiled code, so it is interesting to consider performance ignoring that feature. Without that feature, no attacks are missed and four false-positives occur using a feature for the password or shadow password files, features for C, a feature for logging into the current host, a feature for creating a link, a feature for making a file set user/group id, and a feature for cleaning up after the attack is complete.

For C code, 19/20 attacks can be detected just using the comment and embedded assembly code features, with 6 false positives (out of 1747 files), again training and testing with different data sets. If the comment feature is again not included, the best result comes from scanning for embedded assembly instructions--the addition of no other feature further reduces the error rate for this data set. In this case 15/20 attacks can be detected with four false positives. This feature was encoded using two different approaches; in the first, we simply scanned for gcc's `__asm__` directive; while in the second, we looked for strings of hexadecimal or octal characters, or arrays of hexadecimal or octal values. In many cases, attackers used assembly code to acquire a stack pointer to determine where to copy buffer overflow code, so both features usually co-occur. For the normal data in this study, the `__asm__` feature yielded the best results. False alarms occurred in system-dependent exception handling for the emacs and gdb programs.

As a result of these experiments, it is clear that most attacks can be detected from source code alone, although attacker comments are a valuable feature for detecting software downloaded over a network. If this feature is not used, then other features can be used to accurately detect attack code amongst regular source code. For C code, the best feature is one that is suprisingly easy to detect: simply scan for embedded assembly instructions. Unfortunately, this feature may not be easy to detect in compiled code, and hackers can encode embedded assembly code in ways that are not detected by this feature. Future work will expand the number of attack examples and examine features that appear in compiled code as well as in source code.

There are a number of interesting ways to deploy this technique as it is. In a network-based intrusion detection system, ftp-data, mail, and web transfers can be monitored for the inclusion of attacks. On a host, a process could periodically be run to scan and score source code stored on the disk, or incoming traffic could be examined. This could detect attacks launched locally or compiled locally and launched remotely.

We have shown that a few simple features can differentiate current C and shell attack source code that increases user privileges from normal source code. Simple features such as C code with embedded assembly, or shell code with local log-ins or file ownership changes, and either with the words exploit or sploit or vulnerability embedded in comments result in high detection and low false alarm rates.