Detecting Privilege-Escalating Executable Exploits

Jesse C. Rabek, Robert K. Cunningham, and Roger I. Khazan
MIT Lincoln Laboratory
{rkc, rkh}@ll.mit.edu

Abstract[1]

The Lincoln Laboratory Malicious Code Detector (LIMACODE) is a system for statically detecting privilege-escalating exploits in data streams, such as files and network traffic. LIMACODE operates as follows: it scans data streams, identifies the language of the stream, then extracts language-specific features for input to a feed-forward neural network classifier which labels the stream as either malicious or benign. LIMACODE is designed to be a relatively lightweight system that can classify a large number of streams quickly so as to be deployed at sites where new data streams (e.g., software) appear frequently. This paper describes a part of LIMACODE that detects privilege-escalating exploits embedded in UNIX Executable and Linking Format (ELF) files; the detectors for C and shell code exploits were described earlier elsewhere.

1. Introduction

There are many routes an attacker may take when attempting to compromise a computer system, including employing social engineering, exploiting a vulnerability in a network or local service, or tampering with a physically accessible computer system. However, the damage caused, information gained, resources obtained, etc. is limited by the privileges held by the attacker. On UNIX-based systems, such privileges allow a user to access only those resources that have been specifically granted to the user, to the user's groups, or to the programs that the user is allowed to use [1].

Often attackers attempt to increase the privileges that they hold in order to obtain access to resources that are otherwise unavailable to them. In order to increase their privileges, attackers must either cause the operating system to grant them unauthorized privileges or cause a process with a different set of privileges to perform unauthorized actions on their behalf. Unauthorized privileges can be granted if a system is configured in an insecure manner or if users select weak passwords; attacks against these vulnerability classes are not the focus of this paper. Unauthorized actions can be performed if an attacker can inject code and cause the privileged process to run it. Our focus is this latter case. Specifically, we wish to detect exploits used to perform code injection via out-of-bounds writes, also known as buffer overflows. Buffer overflow vulnerabilities have been and remain one of the most common: between July 2002 and July 2003, 231 out of 712 (approximately 32%) of high severity vulnerabilities published by NIST were from buffer overflows [2].

This paper describes The Lincoln Laboratory Malicious Code Detector (LIMACODE). LIMACODE can detect privilege-escalating C, shell, and executable code. Detectors for source code analysis of attacks that exploit buffer overflows and time-of-check-to-time-of-use errors [3] in privilege escalating C and Shell code are described elsewhere [4]. This paper describes a detector for attacks that exploit buffer overflows in privilege escalating code appearing in Executable and Linking Format (ELF) files compiled for the x86 architecture. ELF is the most common binary executable format used in Linux and Solaris OSs, and x86 processors are the most prevalent. However, there is nothing inherent in our approach that would prevent it from being used on other file system formats, operating systems, or architectures.

2. Background and Related Work

Buffer overflow attacks can be detected using dynamic or static analysis. Dynamic analysis monitors software that is executing, and therefore requires an appropriate environment for running the exploit and vulnerable software, and obtaining information about their interactions. The requisite environment includes the correct operating system, libraries and external programs and an audit system to report on the executing software. In contrast, static techniques examine code without running it. Such techniques are useful in a network in which the ingress of all software is to be monitored, but

[1] This research was sponsored by the Defense Advanced Research Project Agency (DARPA) under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

the software is arriving through various methods (e.g. downloaded in a web browser or pulled from the local ftp server to which outside users upload files) for different operating environments. While the software can be examined as it arrives, the host on which it is arriving or passing through may not be able to run the software for various reasons (e.g. wrong operating system, architecture, or available libraries). Static analysis is also useful for forensic analysis. It is important for our system to be able to detect old attacks, variations on old attacks, and novel attacks. There are several different static analysis approaches that we considered pursuing to meet these requirements. A signature scanner [5] is fast, but does not reliably handle variations on old attacks even when the signature language is highly expressive. Some commercial virus detection systems using this approach are unable to handle even modest code obfuscation [6]. Neither a policy enforcement approach [7] nor an emulation approach were selected because of the computational overhead [8].

Instead, we pursued a machine learning approach to achieve accurate detection, initially examining source code because it was easier to perform feature extraction [9]. Others have pursued similar strategies with binary data, although most examined detection of viruses on DOS and Windows systems. Feature extraction must be performed to achieve fast, accurate classification of malicious software. The amount of intelligence built into the preprocessing step ranges from none, where software is treated as byte sequences with different likelihoods of being in malicious software [10], to some, where a feature is either a byte sequence, a string or a dynamically linked library [11], to substantial, where software is converted into an abstraction pattern prior to matching [6]. In our approach, a few instructions are interpreted and machine learning combines these to create an accurate system that is moderately robust to obfuscating transformations, but which can quickly process new files.

3. System Overview

LIMACODE is a static analysis system that detects old attacks, variations on old attacks, and novel attacks. It uses a language-specific static feature extractor with a feed-forward neural network classifier since this type of system is able to define general classes of privilege escalating attacks and is therefore more robust in determining both novel attacks and variations on old attacks.

The remainder of this section provides a system overview to LIMACODE. A high level flow diagram of the detection process appears in Figure 1.

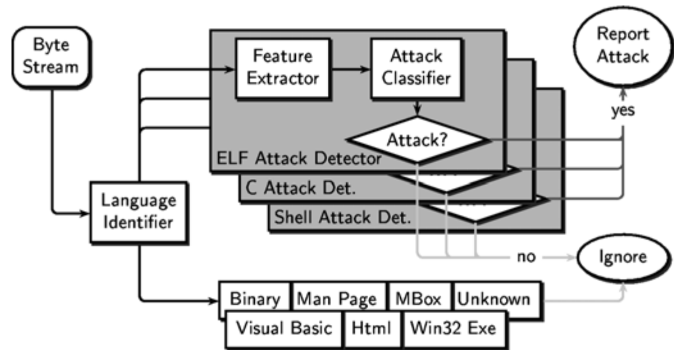

Figure 1. LIMACODE flow chart

The diagram depicts a byte stream (e.g. from a file or network packets) being fed into a language identifier. The language identifier determines which detector (language-specific feature extractor and attack classifier), if any, should be used to analyze the byte stream. The chosen feature extractor examines the byte stream for interesting features and reports them in the form of a vector of integers and real numbers; the vector is then fed into the attack classifier. The output of the attack classifier is the posterior probability that the byte stream does or does not contain privilege-escalating code.

Using this approach, our system is easily extensible to new attack vectors, such as those that employ scripting language formats, by adding more feature extractors and attack classifiers, and then updating the language identifier to include the new byte stream type.

The ELF Attack Detector discussed in this paper operates on ELF files; hence, in the remainder of the paper the byte stream source is assumed to be a file.

3.1. Language Identification

The first step in processing a given file is to identify its language type so that the appropriate feature extractor and attack classifier are used. The language classifier allows each file to only be processed by one detector, thereby speeding the overall detection process at the possible cost of a missed detection. LIMACODE uses a rule-based system that exploits a language's defined structure and syntax to determine a file's type. For ELF files, the language identifier looks for the presence of the ELF magic number at a fixed offset into the file. This simple approach has correctly identifying every ELF file we have encountered. For C and shell source code, the language identifier parses the contents of the file; however, this was not required for ELF files [4].

3.2. Attack Feature Extraction

In order to understand the privilege-escalating attack features and the extraction process, some background on ELF binaries is necessary. ELF binaries consist of a required header section and one or more optional sections [12]. The header section contains information such as the version, target architecture, and the virtual memory address at which the ELF file is loaded. The remaining sections are optional although there are some, such as the *.text* and *.data* sections, that are almost always present. The *.text* section contains the actual executable code, the *.data* section contains initialized data, and the *.rodata* section contains the read-only data.

The ELF feature extractor parses an ELF file into its *.text* and .(*ro*)*data* sections, analyzes these sections, and produces statistics that could identify a given file as malicious. Attack feature statistics represent the steps necessary for injecting code into one or more buffer overflow vulnerabilities. To achieve the best performance on the widest variety of attack code, each feature should encode all possible ways to accomplish a particular part of the attack.

We started with a large number of features in various groups, some inspired from the best features in the LIMACODE C and shell classifiers and some based on our knowledge of how privilege-escalating attacks work. We then used a forward-and-backward, leave-one-out selection process [13] to select those sets of features that best divided the sample space. The selected features can be categorized as a kernel call, instructions in data, or other, and are shown in Table 1.

Table 1: Features used to classify privilege-escalating code.

Group	Name	Count Type	Description
Kernel Call	Exec	N	*Exec* family of functions
	System	N	*System* family of functions
Instructions in Data	Payload	M	Instruction sequences or combinations typically found in injectable buffers
Other	Code in Strings	P	C or shell source code present in strings in the *.data* sections
	Size Data	M	Size of the largest *.data* section

Count Legend: (P)resent, (N)ormalized, (M)aximum.

There are several ways the features statistics are measured: *Present, Normalized, and Maximum*. *Present* indicates if a sample does or does not have a feature. *Normalized* is the number of times a feature appears in a sample normalized by an appropriate divisor to obtain a notion of the density of the feature. *Maximum* records the value of the window with the highest score. All possible combinations were considered during feature selection and the best method for counting a feature is presented in the *Count* column in Table 1.

The remainder of this section describes the observations that led to creating these features and the specifics of our implementations.

Kernel Calls

Observation: Privilege escalating code needs to pass information to a higher privilege process in order to exploit it. Sometimes this is done by starting a vulnerable program with carefully selected arguments. Empirical tests indicate that most benign binaries have a low density of calls to the program initiating services *exec* and *system*. Shorter exploit programs that launch a vulnerable, higher privilege application have a higher density of these calls. We imagine the same is true for inter-process communication calls as well, although we were unable to gather enough training and test samples to verify this.

Implementation: Independently decode and count the occurrences of process creation using the `exec` family of functions[1] that fully specify program path and input, and also the `system` call that uses the shell to specify the environment and determine the absolute program path.

Extension: Interprocess communication (`pipe/signal/shmat/connect`) should also be counted.

Instructions in Data

Observation: Certain types of executable actions rarely appear in localized parts of non-exploit *data* sections unless constructed specifically for injection and execution in a higher privilege process. Among these are software that isolates code location, zeros out registers, and includes control flow.

Implementation: The feature extractor examines a fixed-size sliding window of decoded instructions looking for particular instructions and instruction sequences that accomplish actions that are commonly performed by injected code, adding one point for each type found. Each action is counted only once, even if multiple examples of the action occur within the sliding window. The count thus encodes the number of diverse actions present in a window. The window is used to require locality of actions; window sizes of 20, 30, 40…100 were used with the training data and a window size of 50 consecutive

[1] `execl, execle, execlp, execv` and `execvp`

instructions was found to give the best discrimination. Large non-attack binaries may also contain many of these actions in the *.data* section, but they are distributed over the entire section, whereas dedicated exploit code has tended to have more densely packed features. The following paragraphs detail what each action does and why it is important to look for it.

The first action determines the location of code so that once the exploit has occurred and the buffer is executing, it can pass the address of local data as arguments both to functions provided by standard libraries and to the kernel. This is necessary because the exact location where the payload is injected can depend on the version of the victim application and late-bound library load order, among other factors. An example instruction sequence that will accomplish this is a relative `call` to a `pop` instruction. In this sequence, the call instruction causes the address of the next function to be placed on the stack. The immediate `pop` will place that address into a register, which can then be used as a reference point for its local data. If there is a `jmp` instruction that redirects control flow to the initial `call` instruction in this sequence, then two points are added to the total score for this window since this sequence is prevalent in our training data and is more indicative of an injectable buffer.

The next action looks for instructions that cause control flow changes, since injected code makes decisions based on return codes, and often loops to perform various actions (e.g. file scanning or denying access to some service). LIMACODE looks for local calls and jumps that direct control flow to somewhere within the instruction window. One point is awarded if one or more control flow actions are found.

Another common action is setting a register to zero. Empirical tests indicate that instructions and instruction sequences that zero registers are common in the part of the *.data* sections of malicious ELF files that contain an injectable buffer. There are many ways to do this; for this system, we only include common, single-instruction instructions that accomplish this: *Xor register register; mov register, 0;* and *imul register, 0*. One point is awarded if one or more instructions are found that set a register to zero, regardless of the register used.

The final code-in-data action identifies the presence of one or more kernel calls since injected code frequently needs to interact with the operating system. On x86 based Linux operating systems, this instruction is INT 0x80 (interrupt 0x80). This instruction causes a transition into the kernel from where the call is handled.

In regards to the *Payload* feature as a whole, we found that such a heuristic was necessary since simpler schemes, such as looking for runs of valid instructions, did not work because the IA-32 instruction set is very dense and a random sequence of bytes has a high probability of being a valid sequence of instructions. One of the features that was eliminated as a result of feature selection involved the detection of a payload by identifying valid sequences of instructions in the *.data* section.

Extensions: Other instructions that should also be counted include multiple-instruction sequences that zero registers as well as instructions that reference environment string memory locations.

Other

This final group of features also indicates that the sample attempts to increase privileges, but did not fit into the other two classes.

Observation: It is rare for non-exploit code to embed C or shell code in the *.data* section of an executable; in contrast, exploit code often includes shell code as part of its launching or exploitation process. Also, sometimes C source of an exploit is included in attack software so that it can be compiled differently based on the exact details of the victim application (e.g. version or configuration).

Implementation: In order to detect both C and shell code appearing in strings, all strings appearing in the *.data* section are used as byte stream inputs to a modified versions of the C and shell classifiers respectively [9].

Observation: The vast majority of our training samples had small *.data* sections. This is due to the fact that the exploit code we used for training contained an injectable buffer, the name of the vulnerable program, and little else. Compiler options may dramatically affect the size of the *.text* sections (e.g. static versus dynamically linked), but not the *.data* sections. While not a good indicator on its own, when combined with the other features it significantly improves the ability of the classifier.

Implementation: During the ELF parsing process, the size of the largest *.data* section is recorded and included as part of the feature vector.

3.3. Attack Classification

The attack classifier's neural network is a multi-layer perceptron classifier with a single hidden layer trained using back-propagation of errors [14]. A gradient descent method with a squared error cost function is used for training in which the new weights propagating backward through the network. Training time is negligible using these techniques on the feature vectors and sample sizes used here. Other machine learning techniques were informally explored, but this approach gave the best results for the cases considered.

Prior probabilities of the attack and training classes were equalized to maximize detection rate at the cost of an increased error rate, since more files are benign than malicious. LIMACODE is therefore more likely to

misclassify a benign file as malicious than to misclassify a malicious file as benign.

4. Data Sources

Benign and privilege escalating samples were collected at two different times in order to test the ability of the system to detect new, unseen attacks.

For the malicious samples, C source code was obtained from various websites. Only code that had privilege escalating intent was collected. There are 221 training samples that were collected between July 2001 and January 2002 and included attacks that were released from before that time period. The 68 testing samples were collected between January 2002 and September 2002 and only included samples released in that time period. Test samples were verified as distinct from training samples by performing a byte level comparison of each compiled test sample against the existing compiled training samples. The malicious C files were compiled in as similar a method as possible to the benign samples so as not to leave compilation artifacts that could easily identify the malicious samples.

The benign samples were taken from the /usr/bin directory of a default Red Hat 7.1 installation. The 1280 total benign files were portioned into training and test sets: 979 were randomly chosen to be in the training set and the remaining 301 were used in the test set, to match the ratio of the training and test set of the malicious data.

5. Evaluation

This section presents the ability of LIMACODE to detect new, unseen privilege-escalating ELF binaries and its data processing speed.

5.1. Detection

Figure 23 displays the accuracy of LIMACODE in the form of a detection error tradeoff (DET) curve [15]. In the figure, the false alarm percentage is plotted against the miss percentage for various operating points (i.e. thresholds applied to the output of the classifier). The axes are scaled by normal probability deviates to magnify the target zone.

Unlike fixed-heuristic or signature-based systems, LIMACODE can be operated over a range of operating values, with the operating point selected by a user who specifies the relative value of misses and false alarms. Three points are of particular interest for different uses: the point where the false alarm rate approaches zero, the equal error rate, and the point where the miss rate approaches zero. The first point is interesting for virus detection-like applications where the user wants some protection but mostly does not want other applications to be erroneously labeled. Here, the false alarm rate approaches zero when the miss rate is approximately 30%. Next, the equal error rate describes the point at which miss and false alarm rates are equally important; for LIMACODE's Malicious ELF detector the rate is 4.65%. The final point of interest is one which might be used to scan a captured disk, and for which missing an exploit is much worse then spending the time to examine a false alarm. The miss rate approaches zero when the false alarm rate is about 35%.

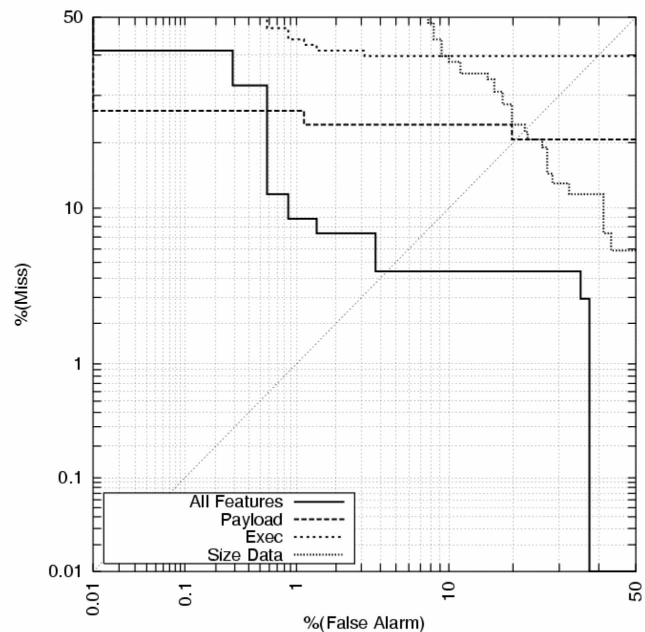

Figure 2: LIMACODE Performance

The results of training and testing the classifier on three different single features are also presented alongside the primary result, since it is conceivable that single features dominate the output. As is clear from the figure, combining multiple features significantly improves the accuracy of the system over most of its operating range. The curves for the accuracy of the isolated System and Code in Strings features do not appear on the graph as they lie outside the region in view. Although individually inaccurate, the integrated system accuracy improves when these features are included.

For false alarm rates less than about 0.6%, the single payload feature is a better discriminator than the classifier that uses all the features. At these low rates, features other than the payload feature introduce a significant amount of noise.

5.2. Throughput

In evaluating the throughput of the system, LIMACODE was used to classify the contents of the /usr/bin directory on a RedHat 9.0 installation. On an Intel Pentium III running at 800 MHz it classified 2,336 files with a total size of 170 MB in 142 seconds or 1.17 MB/s. During execution, LIMACODE spends the vast majority of its time in the feature extraction phase.

6. Discussion

The LIMACODE system does not use signature matching, and is therefore able to detect attacks that it has not seen before. Instead of looking for specific sequences of instructions, features representing actions that are required to exploit a vulnerability in another process encode the fact that there are multiple ways to accomplish the same goal. Our implementation requires locality of multiple required actions. For an attacker to hide from the system, he must obfuscate multiple actions. This is harder to do than to change an isolated signature.

Our approach is resilient to many common obfuscation techniques [6]. It is not affected by code transposition (in which instruction order of an attack is altered), because there is no explicit model of instruction sequences. It is not affected by register reassignment (in which the specific registers used by an attack is changed), because there is no explicit model of a given register for a given attack. It is relatively insensitive to instruction substitution, because in the case where instruction classes are used, all equivalent single-instruction cases are included. (It remains possible to use multi-instruction code to accomplish similar ends, and this is not yet addressed.) Finally, the system is insensitive to dead-code insertion, provided the amount of dead code injected does not cause the exploit to become longer than the scanning window.

There are a number of features that could be added to LIMACODE to increase its accuracy. First, the *Payload* feature could be extended to include more injected buffer actions such as identifying references to environment variable locations, decryptor blocks for obfuscated payloads, and typical API usage sequences. Also, the C source feature set identified calls to *link* as part of the *System Call* group. This feature helped detect exploits of race conditions. However, due to an insufficient number of samples in this newer data set, it was not included in the executable feature set. Collecting a sufficient number of such samples with which to train and test would allow LIMACODE to be tuned to detect this class of privilege escalating attacks.

It is, however, possible to bypass LIMACODE, by causing the counts of the features to change. An attacker could increase the feature count by adding unexecuted dead code to an exploit. Other techniques will work equally well. To respond to this, our system would need to either remove dead code or, equivalently, increase the window size in the presence of dead code. Alternatively, an attacker can decrease the feature count, perhaps by encrypting or obfuscating actions, encoding an action using an instruction or instruction sequence that we don't count, or by spreading out the actions in the *.data* section so that they fall outside of the code window.

Finally, an attacker can avoid the system altogether. Most modern UNIX-like operating systems have compatibility modes, and can execute the older a.out file format (as well as several others). While there is nothing to prevent us from adding support for these file formats, we have not done so.

7. Summary of Results

The most important result from this paper is that it is possible to build an accurate detector of unobfuscated ELF attack code by identifying the specific actions that privilege-escalating code must take in order to accomplish its goal and then detecting code which accomplishes these actions.

LIMACODE raises the skill level required for creating and transmitting an exploit into an enclave. We found that much of the easily obtainable privilege escalating code does not attempt to hide its intent. Therefore, in order to compromise a system protected by LIMACODE, an attacker would have to find or develop intentionally obfuscated attack code.

8. Acknowledgements

We would like to thank Craig Stevenson for his work on the C and Shell code part of the LIMACODE project [4] and his contributions to an early version of the malicious ELF file detector.

References

[1] S. Garfinkel and E. H. Spafford, *Practical Unix and Internet Security*, 2nd ed: O'Reilly & Associates, Inc., 1996.
[2] NIST, "ICAT Metabase," 2000.
[3] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems*, vol. 9, pp. 131-152, 1996.
[4] C. S. Stevenson and R. K. Cunningham, "Accurately Detecting Source Code of Attacks that Increase Privilege," presented at Recent Advances in Intrusion Detection, Davis, CA, 2001.
[5] S. Kumar and E. H. Spafford, "A Generic Virus Scanner in C++," Purdue University, West Lafayette, IN, Technical Report September 17 1992.

[6] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," presented at 12th USENIX Security Symposium, Washington, D.C., 2003.
[7] M. D. J. Bergeron, J. Desharnais, M. M. Erhioui, Y. Lavoie and N. Tawbi, "Static Detection of Malicious Code in Executable Programs," presented at Symposium on Requirements Engineering for Information Security, Indianapolis, Indiana, USA, 2001.
[8] F. Leitold, "Reductions of the general virus detection problem," presented at EICAR International Conference, 2001.
[9] R. K. Cunningham and C. Stevenson, "Accurately Detecting Source Code of Attacks that Increase Privilege," presented at Recent Advances in Intrusion Detection, Davis, CA, 2001.
[10] J. O. Kephart and W. C. Arnold, "Automatic Extraction of Computer Virus Signatures," presented at 4th Annual Virus Bulletin International Conference, Abingdon, England, 1994.
[11] M. G. Schultz, E. Eskin, E. Zadok, M. Bhattacharyya, and S. J. Stolfo, "MEF: Malicious Email Filter," presented at 2001 USENIX Annual Technical Conference, Boston, MA, 2001.
[12] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Standard, Version 1.2," 1995.
[13] *Feature Extraction Construction and Selection: A Data Mining Perspective*: Kluwer International, 1998.
[14] R. P. Lippmann, L. C. Kukolich, and E. Singer, "LNKnet: Neural Network, Machine Learning, and Statistical Software for Pattern Classification," *Lincoln Laboratory Journal*, vol. 6, pp. 249-268, 1993.
[15] A. Martin, G. Doddington, T. Kamm, M. Ordowski, and M. Przybocki, "The DET Curve in Assessment of Detection Task Performance," presented at Eurospeech97, Rhodes, Greece, 1997.