Gaining and Maintaining Confidence in Operating Systems Security

Trent Jaeger Antony Edwards Xiaolan Zhang
IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532 USA
Contact Email: {jaegert}@us.ibm.com

March 15, 2002

1 Introduction

Recently, there has been a lot of work in the verification of security properties in programs. Engler et al. use static analysis to find flaws in the implementation of Linux device drivers, such as the failure to release locks [4]. Edwards et al. use static and dynamic analysis to verify that the authorization hooks of the Linux Security Modules (LSM) framework are placed such that all the necessary authorizations are performed [2, 12]. In addition, Shankar et al. and Larochelle et al. show how to use static analysis tools to find program vulnerabilities, such as buffer overflows and printf vulnerabilities [7, 10, 11]. Lastly, Necula et al. show that we use detect and leverage the cases in which C is used in a type-safe manner in order to detect memory errors [9]. Runtime verification can be used to detect errors in other cases.

While these tools demonstrate that we can use automated tools to gain some level of assurance for our programs, these tools are aimed at individual errors (e.g., static analysis buffer overflows), require significant skill to use properly (e.g., require ad hoc analysis code to find particular driver errors or to write exploits that identify real problems), and are limited to finding only some types of errors in their class (e.g., static analysis is limited by the type safety of C). Ultimately, the goal is to gain a satisfactory level of assurance of the security of the entire program and maintain that level of assurance over the program's evolution (ideally, to improve its level of assurance). Given the breadth and depth of verification approaches, we believe that it is time to examine how they can be put together into a coherent approach for program security assurance.

We contrast the approach of performing assurance by a concerted application of verification tools with the traditional approach to establishing operating system security assurance. The Orange Book and its successor, the Common Criteria, achieve assurance by requiring detailed documentation and design for security from the onset of the project [8, 6]. This security focus is to then guide the transformation of these specifications into an implementation. Some testing and code review is done, but these are ad hoc processes. Further, significant modifications to an assured system are not permitted because they would demand a new labor-intensive assurance, and this approach cannot be applied to already-constructed systems. Not surprisingly given the effort and cost of this form of assurance, very few systems have been built to the higher levels of assurance.

In this paper, we examine some of these verification approaches and develop an approach to operating system security assurance based on automated verification tools. The aim of the approach to enable: (1) practical verification of key security properties; (2) extension in both the types and number of verification tools to improve the breadth and quality of verification over time; and (3) management of verification state to enable maintenance of assurance as the system evolves. We first examine some verification problems in Section 2 to identify the types of tasks that need to be performed in verification. We then examine how individual verification tasks are performed now in Section 3. Then, we propose an approach for performing the necessary analyses and discuss how current analyses may be extended to make this approach work in Section 4.

2 Security Verification Problem

The history of program correctness verification does not generally have a good reputation, but a significant set of program analysis have been constructed that yield practical results. In general, these tools typically perform focused analysis for particular program properties (e.g., Y2K bugs [3], null pointers [5], etc.).

As a result of this experience, we do not believe that security verification is a single process, but rather, a series of analyses that yield greater confidence in the security offered by a program. Also, as new vulnerabilities are discovered, new analyses must be added to the verification toolset in order to maintain confidence in the program. Further, such verification tools should enable practical regression testing of security properties as the program evolves.

For verification that the Linux kernel enforces a system access control policy correctly (i.e., is a correct reference monitor), we identify the following criteria:

- **Verify code integrity:** Detect vulnerabilities that would enable an attacker to run their attack code instead of program code (e.g., buffer overflows, printf vulnerabilities, etc.).
- **Verify complete authorization:** Verify that all controlled operations are executed only after all the necessary authorization requirements are checked. Note that whether permissions to perform these operations are granted to a principal is a policy question, so policy verification is a separate, necessary task that we do not discuss further here.
- **Verify access using authorized object only:** Detect vulnerabilities where an attacker can switch the object used in controlled operations between the authorization and use (i.e., time-of-check-to-time-of-use or TOCTTOU attacks [1]).
- **Detect permission leakage:** Verify enforcement of system invariants necessary to prevent permission leakage, such as closing the necessary descriptors on an `exec`.

If these four criteria are verified effectively, then only the operations authorized by the system's access control policy can be performed. The kernel integrity is protected, all operations are correctly authorized, all accesses use authorized objects, and there are no errors that leak permissions. Thus, relative to the confidence we have in the verification of these criteria, the Linux kernel is assured to enforce the system security policy.

The problem then is to define and implement the analyses that verify these criteria with a reasonable degree of confidence. While perfect static verification of a kernel is impossible given that it is written in C and assembler, a variety of approaches can be employed in unison to improve confidence in the assurance. For example, we use static analysis to detect possible TOCTTOU vulnerabilities [1] by identifying any variable used in a controlled operation that has not been authorized since it was last assigned. This analysis identifies variables as potential vulnerabilities even if they are extracted from an authorized object. While this analysis is conservative in a type safe language, it is possible to modify the value of the variable through non-type safe programming. At present, we assume that the kernel developers are trusted, but ultimately, identification of non-type safe code and verification that this code does not access the stack (for local variables) is necessary to assure this property fully. Necula et al.'s work can find where type safety is not preserved, so other analyses can be built to better verify that such an attack is not possible or only possible in places where runtime checks can be inserted.

Further, a conservative analysis means that several false positives may be identified. Currently, these are examined manually, but many of these are really the same situation, such as extraction of inodes from authorized dentries. Some secondary analyses can make such verification practical. Lastly, since such analyses can be rerun as the kernel evolves, the assurance of the kernel is maintained even as the kernel changes.

3 Verification Experiences

In this section, we outline two example analyses to demonstrate the types of verification tasks that can be done and their effectiveness [1].

3.1 Linux Security Module Verification

The goal of Linux Security Modules (LSM) verification is to ensure that any security-sensitive operation in the Linux kernel is authorized for its proper requirements via the LSM authorization hooks. Since access to such operations involves access to high-level kernel data objects (e.g., inodes, sockets, etc.), we identify any access to these data structures as a mediation point to be authorized (i.e., a *controlled operation*).

We use a runtime analysis tool to identify authorization requirements for the controlled operations and anomalies to those requirements []. To start, we assume that LSM is largely correct, so we can express invariants that indicate an anomaly in the authorization. An example of an anomaly is a controlled operation that has different authorizations for different runs of the same system call. The other controlled operations are classified as consistent, but since we may be missing an authorization entirely there can still be an error. Since all controlled operations usually require the same authorizations, these

[1] We would expect to expand this section for the full paper, so more experiences can be analyzed.

errors are easy to identify. For each anomaly, we must determine if there is an exploitable situation. At present, we have found 5 significant anomalies, where the LSM community agreed that 4 are truly errors and the other case works under the limited circumstances intended.

Runtime analysis is limited by the statement coverage and input value coverage provided by the benchmarks. In the first case, we have found that performance benchmarks only execute 20% of the Linux kernel statements. Further, the ability to leverage a possible TOCTTOU situation requires active attacks to provide the inputs necessary for the analysis to see the changed value. Therefore, we also use static analysis techniques to find cases where controlled operations are performed using variables that are not authorized as expected. Most of these cases turn out to be possible TOCTTOU vulnerabilities where a variable is authorized, but it is recomputed from higher-level objects rather than used directly. Again, exploits must be written to determine whether such situations are vulnerabilities or not.

When the kernel is modified, we can verify that the authorization requirements and no TOCTTOU vulnerabilities have been added. Modifications to files and functions indicate the scope of regression testing. In general, all the system calls that have code paths that intersect the modified code may need to be tested, although some optimization are possible for static analysis. For example, when no new controlled operation variables or code paths are introduced and we can see that the order between the authorization and the operation is maintained then verification is not necessary.

3.2 Buffer Overflow Detection

Wagner, et al. have developed a static analysis tool to detect potential buffer overflow vulnerabilities in C code [11].

In their approach, C strings (character arrays) are modeled as an abstract data type manipulated via the standard C library functions (e.g. strcpy, strcat, sprintf). Therefore, buffer overflows caused by manipulating strings directly cannot be detected, however, they claim that this represents only a small portion of the vulnerabilities.

Each string in the program is associated with two ranges. One range stores the number of bytes allocated to the string, the other stores the number of bytes in use. C string functions are modeled by their effect on these ranges. Each integer variable in the program is also associated with a range of possible values. The tool then performs a, flow-insensitive, integer range analysis that maintains these ranges, and checks for violations of the safety property: `in_use(s)` $\leq$ `alloc(s)`.

Flow-insensitivity was chosen to allow efficient analysis of large programs. Unfortunately, it also leads to a large number of false-positives that must be manually inspected by a human. Several vulnerabilities were found, and they also identified some that were found not to be exploitable.

4 Verification Approach

Assurance consists of running analyses to verify the four types of security properties. The first and fourth security properties are ad hoc, so multiple analyses may be performed for each. For example, buffer overflow and printf vulnerabilities involve different analyses.

In general, each analysis consists of the following:

- **Scope:** Each analysis works under a possibly null set of assumptions (e.g., type safety). Obviously, the fewer assumptions an analysis depends on, the broader its scope.
- **Scope verification:** Therefore, other analyses may be necessary to maximize the likelihood that all assumptions hold.
- **Classifications:** Each analysis classifies the relevant cases (e.g., positive and negative). Most analyses have some false positives, but a good analysis will have a manageable set of false positives and no false negatives.
- **Classification analysis:** Subsequent analyses may be necessary to verify that a classification is correct with respect to the security requirements (e.g., deriving and running potential exploit programs for positive cases).
- **Case dependencies:** Each analysis result depends on some conditions that, if unchanged, do not require the regression testing of a case upon system modification.

First, each analysis may depend on certain assumptions. For LSM verification, we assume that all controlled operations are performed on variables of controlled data types. Thus, the analysis can handle a variety of bizarre type castings, but cannot detect accesses through other data types, such as `char *`. Initially, we assume that kernel developers are trusted not to do such things, but we would ultimately like to leverage Necula et al.'s approach to protect against non-type safe code [9]. These

analyses will likely have some probabilistic nature and be highly domain-dependent. For example, for a particular function that is not type-safe, we may want to detect whether it can ever access particular controlled data objects.

The main goal of each analysis is to classify its cases into positive (i.e., likely errors) and negative (i.e., likely correct) cases. For all but the simplest analyses, false classifications are possible. In LSM verification, we are conservative about our static identification of positives and do not generate any false negatives (under our analysis scope). This is not the case with the buffer overflow detection where a small number of false negatives are permitted. Then, we have to perform subsequent analyses on positives. Currently, such analyses consist of manual inspection and exploit generation in both tools. For LSM verification, some manual inspections, such as initialization functions and extraction of inodes from checked dentries, should be automated easily. Further, we envision moving to templates for exploit generation, similar to using templates to describe attacks [1]. We would identify system calls that enable modification of relationships (e.g., descriptor to file via `dup`) and means for triggering reschedules (e.g., forcing page faults) to implement TOCTTOU attacks.

Thus, the analysis results in classifications into positives, negatives, and false cases of each. As the kernel is modified, we would like to enable system regression testing commensurate with the extent of the changes. Minimizing the effort involves eliminating the cases where the factors that determined its classification are not changed. There are two sets of factors. First, all classifications depend on the execution context in which the case is run. For LSM verification, each controlled operation is run in a system call path and is authorized by certain LSM hooks, so as long as these remain fixed re-verification is not necessary. Second, the reasons that cases are falsely classified determine whether they will be again. For LSM verification, initialization cases can be easily identified, and we can verify that an association between dentries and inodes is still permanent (e.g., because the inode field in the dentry is never reset).

5 Conclusions

We develop an approach for operating system assurance, in particular the Linux kernel's ability to serve as a correct reference monitor, extrapolated from current research in security property verification. We find such analyses currently enable identification of vulnerabilities, but work under a variety of assumptions and require significant effort to use. We propose an approach whereby assumptions are justified, a sequence of analyses enable complete classification of cases, and regression testing is possible. The approach enables further development of analyses into a coherent framework, so the system's assurance can be determined with confidence, this confidence can be enhanced with improved analyses, and this confidence can be maintained when the system evolves.

References

[1] M. Bishop and M. Dilger. Checking for race conditions in file accesses. Technical Report CSE-95-10, University of California at Davis, September 1995.

[2] A. Edwards, T. Jaeger, and X. Zhang. Verifying authorization hook placement for the Linux Security Modules framework. TR 22254, IBM, December 2001.

[3] M. Elsman, J. S. Foster, and A. Aiken. Carillon – a system to find Y2K problems in C programs, user manual. www.cs.berkeley.edu/carillon, 1999.

[4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operation System Design and Implementation (OSDI)*, October 2000.

[5] D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[6] ITSEC. *Common Criteria for Information Security Technology Evaluation.* ITSEC, 1998. Available at www.commoncriteria.org.

[7] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, 2001.

[8] NCSC. *Trusted Computer Security Evaluation Criteria.* National Computer Security Center, 1985. DoD 5200.28-STD, also known as the Orange Book.

[9] G. Necula, S. McPeak, and W.. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*, 2002.

[10] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the Tenth USENIX Security Symposium*, 2001.

[11] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS Network and Distributed System Security Symposium*, 2000.

[12] X. Zhang, A. Edwards, and T. Jaeger. Using CQual for static analysis of authorization hook placement, February 2002. Submitted for conference publication.