

Systematic Scanning for Malicious Source Code

S. Tucker Taft, SofCheck, Inc., 11 Cypress Drive, Burlington, MA 01803, +1-781-750-8068,
tucker.taft@sofcheck.com

Abstract— *For an organization that depends on software for important parts of its mission, safety and security flaws in such software are major concerns. Although there is a growing number of tools that can be helpful in identifying unintentionally inserted safety or security flaws, the possibility of intentionally inserted “flaws” or back doors can no longer be ignored.*

Fundamentally, an intentionally inserted back door can only be recognized by the fact that it does more rather than less of what it is supposed to do. For example, a function that is expected to only query the balance of a bank account, may also, as a side-effect under special circumstances that are unlikely to be encountered during testing, transfer money between accounts. To locate such hidden side-effects, it requires that the semantics of each function somehow be extracted from the source code and presented to a reviewer in a way that allows them to recognize inappropriate actions.

In this paper we describe scanning technology that can automatically extract the pre- and post-conditions of every function in the system, including both direct and indirect effects of each function, and present these to a reviewer in human-comprehensible terms. For each external entry point into the system, the post-conditions in particular may then be compared against the expected effects of the function, and where potentially inappropriate side-effects are identified, these effects may be traced down through the program to the point where they occur.

Index Terms—*source code analysis, preconditions and post-conditions, malicious code, back doors*

I. INTRODUCTION

THE supply chain for software has grown dramatically more complex in the past decade. Mission-critical software systems are now incorporating commercial off-the-shelf components, components from the open-source community, as well as custom components developed by outsourcing companies, many of which may employ offshore development groups. On the other side, the threats against software have also grown more sophisticated in the past decade, with new classes of threats such as cross-site scripting and SQL injection joining older threats such as buffer overflow and Trojan-horse back doors.

For an organization that depends on software for important parts of its mission, safety and security flaws in such software are major concerns. Although there is a growing number of

tools that can be helpful in identifying unintentionally inserted safety or security flaws, the possibility of intentionally inserted “flaws” or back doors can no longer be ignored. Source-code scanning tools generally look for unsafe or insecure coding practices, but intentionally inserted back doors will very likely avoid all such practices, and in fact appear as well-written code, albeit intentionally open for remote malicious exploitation.

Is it feasible to automate the process of scanning source code for intentionally inserted back doors? Fundamentally such source code can only be recognized by the fact that it does *more* rather than less of what it is supposed to do. For example, a function that is expected to only query the balance of a bank account, may also, as a side-effect under special circumstances that are unlikely to be encountered during testing, transfer money between accounts. To locate such hidden side-effects, it requires that the *semantics* of each function somehow be extracted from the source code and presented to a reviewer in a way that allows them to recognize inappropriate actions.

One way to describe the semantics of a function is in terms of its pre- and postconditions. This paper describes a technology that can be used to automatically extract the pre- and postconditions of every function in a system, including both direct and indirect effects of each function, and present them to a reviewer in human-comprehensible terms [1]. For each external entry point into the system, the postconditions in particular can then be compared against the expected effects of the function, and where potentially inappropriate side-effects are identified, these effects can be traced down through the program to the point where they occur. For a large system, pure manual review of source code for this kind of inappropriate side effect is infeasible, since in general it is not that the action being taken is inherently inappropriate, but rather it is the circumstances under which it is occurring that are incorrect.

II. AUTOMATIC EXTRACTION OF FUNCTIONAL SEMANTICS

The first step in automatically extracting semantics from a software system requires performing a bottom-walk of the call graph of the system, analyzing each function for its inputs, outputs, and new object creations. For the inputs, flow analysis must be performed to determine what sets of input values are handled by the function, and what sets of inputs result in run-time failures or undefined behavior. The sets of input values that are handled by the function can be considered the “preconditions” for the inputs. Presuming the

inputs remain within these acceptable ranges, the sets of possible final values for each output (including newly created objects) then need to be computed by symbolically approximating the net effect of the function, iterating until reaching a fixed-point. These final values represent the “postconditions” of the function. Ideally both the preconditions and postconditions are expressed symbolically rather than numerically where appropriate. For example, a symbolic precondition might be “ $X < A.length$ ” and a symbolic postcondition might be “ $X = \text{old } X + 1$ ”.

For the purposes of identifying malicious code, it is important that the pre- and postconditions capture requirements and effects not only on explicit parameters and results of the function, but also on any global variables. Furthermore, if the function includes calls on other functions (that is, it is not a “leaf” in the call graph), then the pre- and postconditions should cover effects of any function called directly or indirectly by the given function. In the presence of recursion, an iteration over the call graph will generally be necessary to achieve a fixed-point approximation. Ultimately, the overall pre- and postconditions for “entry points” into the system, such as the doGet or doPost operation in a web servlet, are of interest to link external actions to a potentially malicious effect.

Producing comprehensible symbolic pre- and postconditions for every function is a challenge in itself [1], [2], but for the purposes of identifying malicious code, it is also important to factor in the effect of calls to code outside of the set of source code that has been provided for analysis. Calling external subsystems such as a database manager or a network subsystem represent important side effects, and may be critical to identifying suspicious code. For example, a reviewer looking for malicious code would certainly want to know that a function that was thought to do a relatively simple, self-contained computation, under certain circumstances calls a function that might initiate a network transmission. It is also useful to know what use is made of any values returned by a call on an external subsystem. Therefore, as part of reporting on pre- and postconditions, it is may also be useful to report what “presumptions” are being made by the code of a given function about the values of data returned from external calls.

III. HISTORICAL DATABASE

Having automatically extracted and reported on the preconditions, postconditions, references to external subsystems, and presumptions about values returned from such subsystems, the human analyst presumably has enough information to scan for malicious code. However, the total amount of such information can be overwhelming. Even if the analyst focuses only on the overall effects of external entry points into the system, the number of objects manipulated and external subsystems called may still be very large.

In general, most complex systems are not built in a single development cycle, by a single development group. Instead, most complex systems evolve over many years, and go

through a series of development stages. Normally concerns about malicious code do not arise during the initial stages of development performed by a trusted in-house team. However, later stages of development may be farmed out to other development teams, potentially in other companies, or potentially off shore. At this point, there is generally less oversight possible, and the teams may be less trusted or perhaps completely unknown.

To isolate the development performed by an initial, trusted team of developers, from later development performed by a less trusted team, an analysis tool can use an historical database recording the results of its analysis at each stage of development. In particular, before sending a system out for enhancement by a less trusted team, the full set of preconditions, postconditions, external calls, and presumptions on external results can be recorded in the historical database as a baseline. Then when the enhancements are complete, the source code can be scanned again, and the tool can identify just the changes (the “deltas”) between the results and the new results. The analyst can then focus only on parts of the code where there were significant changes, and particularly on places where there were new or changed postconditions, or additional calls on external subsystems. Any changes in postconditions or external calls should be tied to expected enhancements to the system. If there are functions whose behavior was not expected to change, or for which the changes were carefully circumscribed, any unexpected change in postconditions or in sets of external calls would be a “red flag” for the analyst.

TABLE I
EXAMPLE OUTPUT SHOWING ANNOTATION DELTAS

• [TableRow sendInfo\(Context, String, bool, bool\)](#)

-/+	Kind	Annotation Text	Date
-	presumption	create(...) != null	2008-03-01 13:54:18
-	presumption	rd.data@233 != null	2008-03-01 13:54:18
-	presumption	rd.data@245 != null	2008-03-01 13:54:18
+	post	init'ed(new TableRow(create#1).table)	2008-03-01 13:54:18
+	post	new HashMap(TableRow#1) num objects <= 1	2008-03-01 13:54:18
+	post	new TableRow(create#1) num objects <= 1	2008-03-01 13:54:18
+	post	new TableRow(create#1).data == &new HashMap(TableRow#1)	2008-03-01 13:54:18
-	post	org/dspace/core/ConfigurationManager.loadedFile == One-of{old org/dspace/core/ConfigurationManager.loadedFile, &new File(loadConfig#4), &new File(loadConfig#5)}	2008-03-01 13:54:18
-	post	org/dspace/core/ConfigurationManager.properties == One-of{old org/dspace/core/ConfigurationManager.properties, &new Properties(loadConfig#6)}	2008-03-01 13:54:18
+	post	possibly_updated(org/dspace/core/ConfigurationManager.loadedFile)	2008-03-01 13:54:18
-	unanalyzed	call on execute	2008-03-01 13:54:18
+	unanalyzed	call on insert	2008-03-01 13:54:18
-	unanalyzed	call on java.util.Arrays.asList	2008-03-01 13:54:18
-	unanalyzed	call on java.util.List.addAll	2008-03-01 13:54:18

Table I is an example of the annotation “deltas” report for the function “sendInfo” that shows only *changes* to preconditions, presumptions, postconditions, and unanalyzed calls on external functions. The first column shows whether the annotation was dropped (“-”) or added(“+”), the next

shows the kind of annotation, the third column shows the actual annotation, and the last shows the date of the inspection when the “drop” or “add” was first detected. In this example, since the baseline inspection, there were no changes to preconditions, three presumptions were dropped, two postconditions were dropped and five added, and three external (“unanalyzed”) calls were dropped and one added. In this case, the additional call on “insert” might be a flag to the reviewer, if this is seen as a potentially “dangerous” external call. Note that most functions in a large system would have no changes in their annotations given a modest incremental revision, so only those functions with at least some annotation changes need be reviewed. Further, it is possible to tell the tool to show only changes to postconditions and unanalyzed calls, if those are the ones of particular interest for malicious code analysis.

Given the alternative of manually reviewing hundreds of thousands of changed lines of code, having an historical database identifying just the deltas to the postconditions and external calls can make the systematic scanning for malicious code into a manageable task.

IV. CONCLUSION

Given the number of components in a modern software system, the only practical way to find places where an inappropriate action is being taken is to have automated support. However, to be useful, the automated support needs to focus on the overall semantics of a given function, not simply on whether it abides by some number of desired coding practices. As we have described above, a number of challenges exist in both the scanning technology for extracting these semantics from the source code, and presenting them to the user for their systematic review. Nevertheless, we have shown that these challenges can be met, providing a way to systematically identify places where malicious source code might be lurking in even large software systems.

V. REFERENCES

- [1] S. Tucker Taft, *The automatic extraction of semantic information using advanced static analysis*, Ada UK Conference 2007, http://www.adacore.com/multimedia/aa_videos/lectures/adauk07_06_slides.pdf,
- [2] Nikola Milanovic, Miroslaw Malek Extracting Functional and Non-functional Contracts from Java Classes and Enterprise Java Beans, *Proceedings of the Workshop on Architecting Dependable Systems (WADS 2004) at the International Conference on Dependable Systems and Networks (DSN 2004)*, Florence, Italy, 2004.