Poster: A Runtime Approach to Security Auditing

Byron Hawkins, Brian Demsky
University of California, Irvine, USA
{byronh,bdemsky}@uci.edu

Michael B. Taylor
University of California, Irvine, USA
mbtaylor@ucsd.edu

I. Overview

After a software system is compromised, it can be difficult to understand what vulnerabilities attackers exploited. Any information residing on that machine cannot be trusted as attackers may have tampered with it to cover their tracks. Moreover, even after an exploit is known, it can be difficult to determine whether it has been used to compromise a given machine. Aviation has long-used black boxes to better understand the causes of accidents, enabling improvements that reduce the likelihood of future accidents.

Many attacks introduce abnormal control flows to compromise systems. In this poster, we present BlackBox [2], a monitoring system for COTS software. Our techniques enable BlackBox to efficiently monitor unexpected and potentially harmful control flow in COTS binaries. As depicted in Figure 1, BlackBox constructs dynamic profiles of an application's typical control flows to filter the vast majority of expected control flow behavior, leaving us with a manageable amount of data that can be logged across the network to remote devices. BlackBox can also be configured with a control flow blacklist to prevent known and foreseen exploits.

BlackBox has 14.7% overhead on the SPEC CPU 2006 benchmark suite (geometric mean). Experiments demonstrate BlackBox successfully logs and blacklists recent exploits.

II. Monitoring

As BlackBox monitors a program, it continuously constructs a control flow graph (CFG) of the program execution, where a node represents one basic block and an edge represents a control flow transfer between them. Each new node and edge in the graph represent a *program action* that is logged to the remote server. The naïve implementation of BlackBox would (a) perform poorly on the user's desktop and network, (b) flood the log with normal (safe) *program actions*, and (c) report spurious warnings for special cases in program code. Four techniques make it possible for BlackBox to efficiently and effectively monitor popular desktop applications such as Microsoft Office, Google Chrome and Adobe PDF Reader.

A. Log Filtering

Since any delay in the transfer of a log entry to the remote server could potentially allow the adversary to obscure an attack, BlackBox must reduce the total log size enough that each entry can be transferred immediately. BlackBox learns the normal behavior of a program during an offline profiling phase and records the observed CFG to a *trusted profile* that it can use to filter *program actions*: when the monitored program takes any action that is already recorded in the *trusted profile*, logging is elided. BlackBox employs a complete shadow stack to additionally elide all conventional function-return edges. Where a naïve control flow monitor would log millions of unique events per hour for large desktop applications, BlackBox logs less than 100 events per hour, yet consistently records the pivotal actions of exploits.

B. Trusting Unconventional Code

Windows programs commonly overwrite small fragments of their own code at runtime, for example to wrap or replace system calls. This violates the common security practice of making memory either writable or executable, but not both ($W \bigoplus X$). Programs also make unconventional use of the `ret` instruction, for example the soft-thread dispatcher in the Windows `fibers` API writes the fiber start address into the return's stack slot and issues a bogus `ret`. BlackBox adds such anomalies to the *trusted profile* so that these otherwise alarming *program actions* will be filtered from the log.

C. Trusting Dynamically Generated Code

The use of code generators is becoming increasingly popular in desktop software, even in traditional applications such as Microsoft Office. But random factors in JIT engines make dynamically generated code (DGC) incompatible with the *trusted profile* approach, since the low-level construction of DGC differs significantly between observationally identical executions. To avoid flooding the log with *program actions* from ordinary (safe) DGC, BlackBox leverages the insight that DGC can be trusted if (a) it was generated in the normal way by the application's known code generator, and (b) the DGC routines only interact with a trusted set of entry and exit points in the application (similar to RockJIT [3], but supporting COTS applications, and compatible with JITs such as the Microsoft Managed Runtime that dynamically generate linkage functions between modules that were discovered at runtime). Three *program actions* specifically focus on DGC:

- **gencode write** from node A to node B indicates that an instruction in node A wrote dynamic code B.
- **gencode chmod** from A to B indicates that A changed the `executable` permission on a page containing B.
- **gencode call** represents an entry/exit between dynamic code and statically compiled code.

To avoid aliasing in common functions like `memcpy`, an edge is also created from each frame on A's call stack to B.

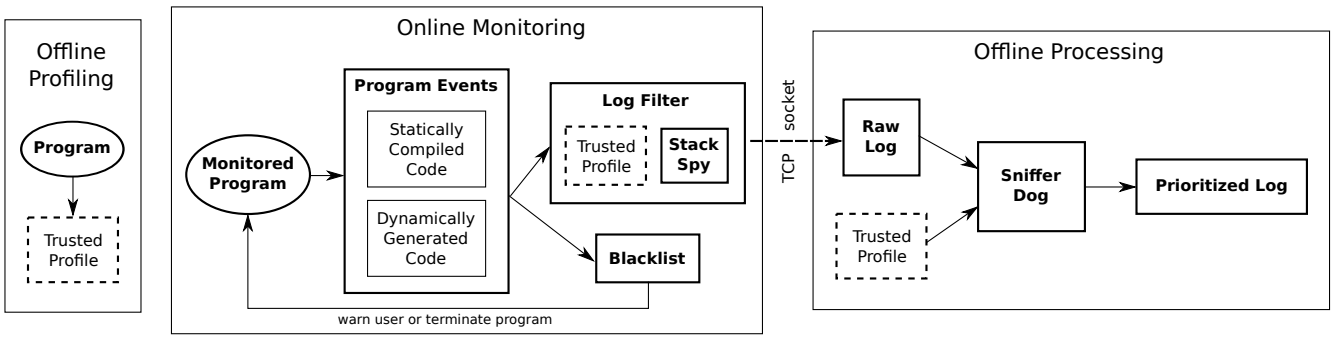

Fig. 1. The three phases of BLACKBOX program monitoring.

In the offline profiling phase, BLACKBOX records these DGC *program actions* to the *trusted profile*. During online monitoring, if the program generates code according to this model, the DGC is trusted and logging is elided—but any deviating *program actions* are logged as a potential code injection. Similarly, any interaction with DGC that does not conform to the *gencode call* edges in the *trusted profile* is logged, even if the DGC was generated in a trusted way.

D. Preventative Monitoring

Analysis of the BLACKBOX logs may often reveal program vulnerabilities and other potential issues before any damage occurs. For example, it may be possible to identify exploitable program errors, or violations to usage policies such as restrictions on installing third-party software or plugins. But the manual process of log analysis is inherently more labor intensive for preventative monitoring than for triage of a specific incident, since the goal is not to find the cause of *one* problem, but to detect *any* foreseeable problem. BLACKBOX facilitates preventative monitoring with two components that highlight the most suspicious program behaviors in the log.

1) Stack Spy: Since the damaging actions of an exploit require the use of system calls, in most cases the adversary will cause at least one deviation in normal control flow en route to the syscalls of the payload. BLACKBOX detects this scenario using a *stack suspicion* flag per thread. The *stack suspicion* flag is activated in a stack frame when an untrusted *program action* first occurs, and remains activated until that stack frame returns. All syscalls that are invoked under *stack suspicion* are logged, even if the syscall site itself is trusted.

2) Sniffer Dog: For large desktop applications, profiling is rarely able to attain complete coverage of normal execution, so the BLACKBOX logs often contain some untrusted-but-benign *program actions*. An offline log sorter called *sniffer dog* employs a principle of "typical irregularities" to estimate the probability that an untrusted *program action* is a safe variant of trusted behavior. The histogram of *trusted profile* training reveals how frequently new edges of each type are normally discovered between each pairing of modules (reflexive included), and *sniffer dog* summarizes each with a PowerLaw model [1]. Log entries conforming to the PowerLaw model are given lower priority, while those exceeding the model's prediction for new events are given higher priority.

III. ANTI-VIRUS 2.0

One of today's most widely deployed security strategies is anti-virus, which relies on a labor-intensive manual analysis by highly skilled experts that aims to develop a comprehensive malware blacklist. The effort begins with *malware diagnosis* to determine the symptoms of an infected binary, usually by executing it in an analysis sandbox. But malware may probe for evidence of such analysis and hide its malicious behaviors, making the process especially difficult. Once the effects of a virus are eventually understood, a *binary signature* is generated to uniquely match infected files on disk. But randomization of malware payloads dramatically increases this effort, since each variant requires a different signature. As malware development tools improve, the cost of the traditional anti-virus approach rises, even as its reliability is eroded.

The BLACKBOX monitoring system provides an alternative to *malware diagnosis* that not only reveals control flow details of exploits as they occurred in the field, it is also immune to evasion tactics—malware has nothing to gain by withholding its payload while under observation, because BLACKBOX monitors the attack target itself (at the end user's site).

BLACKBOX also provides a low-cost, reliable alternative to *binary signature matching*. Instead of attempting to uniquely identify a malicious executable as it appears on disk, BLACKBOX implements a control flow blacklist that blocks an exploit at the specific point it would compromise the monitored program. This approach is immune to superficial randomization because malware has a limited number of opportunities to exploit a given program. Since the BLACKBOX log and blacklist have similar formats, the log analyst can easily transform a log entry into a blacklist entry that prevents the same malicious *program action* from recurring in the future.

BLACKBOX *is supported by the NSF under award* 1228992, *grants* CCF-0846195, CCF-1217854, CNS-1228995, *and* CCF-1319786; *and by the* Center for Future Architectures Research (C-FAR), *one of six centers of* STARnet, *a* Semiconductor Research Corporation *program sponsored by* MARCO *and* DARPA.

REFERENCES

[1] J. Alstott, E. Bullmore, and D. Plenz. powerlaw: A python package for analysis of heavy-tailed distributions. *PLoS ONE*, 9(1):e85777, 2014.
[2] B. Hawkins, B. Demsky, and M. B. Taylor. Blackbox: Lightweight security monitoring for COTS binaries. In *CGO*, 2016.
[3] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *CCS*, 2014.