Detecting Code Reuse Attacks with a Model of Conformant Program Execution

Emily R. Jacobson, Andrew R. Bernat,
William R. Williams, and Barton P. Miller

Computer Sciences Department, University of Wisconsin
{jacobson,bernat,bill,bart}@cs.wisc.edu

Introduction. Code reuse attacks are an increasingly popular technique for circumventing program protection mechanisms. Traditionally, security analysts were concerned with code injection attacks; $W \oplus X$, which marks pages as exclusively writable (W) or executable (X), disallows these attacks. Code *reuse* attacks bypass $W \oplus X$ by constructing exploits from code already present within a process; thus, new security approaches are required.

We present a novel technique for efficient, robust detection of code reuse attacks. Unlike related approaches that rely on an understanding of expected exploit characteristics, our work is grounded in a model of conformant program execution ($\mathcal{CPE}$), in which we define what program states are possible during normal execution. We demonstrate that code reuse attacks violate this model and thus can be detected. We generate our model automatically from the program binary; thus, no learning phase or expert knowledge is required, and new exploit variations will not circumvent $\mathcal{CPE}$. $\mathcal{CPE}$ has high overhead, so we define *observed conformant program execution* ($\mathcal{OCPE}$), which validates program state at system calls. $\mathcal{OCPE}$ imposes low overhead as compared to other techniques; we demonstrate that this relaxed model is sufficient to detect code reuse attacks.

We have implemented our model of $\mathcal{OCPE}$ in a tool, ROPStop. At the core of ROPStop is a strong binary analysis of the code. Unlike previous work, ROPStop does not rely on known attack characteristics and runs on unmodified binaries. In our testing, ROPStop accurately detected real exploits while imposing an average 5.42% overhead on conventional binaries from SPEC CPU2006.

Background. Code reuse attacks search the address space for useful sequences of instructions, *gadgets*, and chain these gadgets together to perform the attack. Return-oriented programming (ROP) uses return instructions to chain together gadgets; jump-oriented programming (JOP) uses indirect jump instructions [2].

There are a variety of existing techniques designed to mitigate or detect code reuse attacks. Mitigation approaches make gadget discovery more difficult via ASLR or software diversification; however, these techniques do not preclude code reuse attacks, but simply challenge attackers to identify gadgets in more sophisticated ways. Existing detection techniques identify expected characteristics of these attacks: e.g., expected gadget composition or size, or frequent returns. In contrast, our work focuses on detecting any violations of $\mathcal{CPE}$ and does not rely on known attack behaviors.

S. Salvatore, S. Angelos, and W. Charles (Eds.): RAID 2013, LNCS 8145, pp. 452–453, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Control flow enforcement (e.g., CFI) and anomalous system call detection (e.g., host-based IDS) may also be effective against code reuse attacks. Unlike CFI, our work can be applied to an unmodified, running process; unlike learning-based IDS, our work is based on a model of $\mathcal{CPE}$. Further, $\mathcal{OCPE}$ enforces valid program state at each system call, rather than a valid pattern of system calls.

Conformant Program Execution. $\mathcal{CPE}$ is based on observable properties of the program counter and runtime callstack. A program P is *conformant* if, for a given program state, the program counter and callstack are individually valid and consistent with each other. P has $\mathcal{CPE}$ if the program is conformant for all program states during the execution of P.

A program counter is valid if it points to an instruction in the set of valid instructions for the program. This requirement eliminates the use of unaligned instructions that could provide a rich selection of unintended instruction sequences to be used in an attack. A callstack C is valid if a height requirement holds for each frame in C and if a call requirement holds for each pair of adjacent frames. Validating calls between procedures associated with consecutive stack frames ensures that C represents a valid control flow path through P.

Implementation. ROPStop uses several components from the Dyninst binary modification and analysis toolkit to perform runtime monitoring and verification [1]. ProcControlAPI creates a new process or attaches to a running process and allows the user (ROPStop) to register callbacks at interesting events; we augmented ProcControlAPI to allow callbacks at system call entry. ParseAPI uses recursive traversal parsing to construct a whole-program control flow graph; this analysis uses sophisticated heuristics to recognize functions that are only reached by indirect control flow and works in the absence of symbol table information. StackwalkerAPI gathers full callstacks; we extended StackwalkerAPI to use static dataflow analysis to calculate stack heights. This robust analysis enables an accurate stackwalk in the absence of debugging information.

Evaluation. We evaluated ROPStop using 4 real ROP and JOP exploits and a stack smashing attack; ROPStop identifies these exploits with 100% accuracy. We tested ROPStop with SPEC CPU2006 as a control group of conventional binaries to evaluate overhead and measure the occurrence of false positives; ROPStop has an average overhead of 5.42% and no false positives.

References

1. Paradyn Project: Dyninst (2012), `http://www.dyninst.org`
2. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-Oriented Programming: Systems, Languages, and Applications. ACM Trans. Info. & System Security 15(1), 2:1–2:34 (Mar 2012)