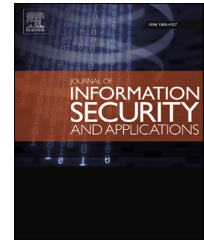
Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/jisa



ROPocop — Dynamic mitigation of code-reuse attacks

Andreas Follner *, *Eric Bodden*

Secure Software Engineering Group, Technische Universität Darmstadt, Rheinstr. 75, 64295 Darmstadt, Germany

ARTICLE INFO

Article history:
Available online 22 February 2016

Keywords:
Buffer overflow
Return-oriented programming
Code-reuse attack
System security
Exploit mitigation
Dynamic binary instrumentation

ABSTRACT

Control-flow attacks, usually achieved by exploiting a buffer-overflow vulnerability, have been a serious threat to system security for over fifteen years. Researchers have answered the threat with various mitigation techniques; but nevertheless, new exploits that successfully bypass these technologies still appear on a regular basis.

In this paper, we propose ROPocop, a novel approach for detecting and preventing the execution of injected code and for mitigating code-reuse attacks such as return-oriented programming (RoP). ROPocop uses dynamic binary instrumentation, requiring neither access to source code nor debug symbols or changes to the operating system. It mitigates attacks both by monitoring the program counter at potentially dangerous points and by detecting suspicious program flows.

We have implemented ROPocop for Windows x86 using PIN, a dynamic program instrumentation framework from Intel. Benchmarks using the SPEC CPU2006 suite show an average overhead of 2.4×, which is comparable to similar approaches, which give weaker guarantees. Real-world applications show only an initially noticeable input lag and no stutter. In our evaluation our tool successfully detected all 11 of the latest real-world code-reuse exploits, with no false alarms. Therefore, despite the overhead, it is a viable, temporary solution to secure critical systems against exploits if a vendor patch is not yet available.

© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

Attacks that aim at manipulating a program's control flow, often through a buffer overflow vulnerability, are still one of the biggest threats to software written in unsafe languages like C or C++ (Bubinas, 2013). If successfully exploited, control-flow attacks can allow an adversary to execute arbitrary code. In the early 2000s, operating-system developers started adding mitigation techniques into their software. To this day, new techniques are added on a regular basis; however, while they make successful and reliable exploitation much more difficult, they can be bypassed.

Contests like, e.g., pwn2own (Gunn, 2014) continuously show that current mitigation techniques are insufficient when it comes to protecting applications, and that more comprehensive methods are required. Currently, the most widely used attack technique, and an essential part of virtually every exploit, is RoP (Roemer et al., 2012), where instead of injecting new code, an attacker pieces together short code fragments, which already

* *Corresponding author.* Secure Software Engineering Group, Technische Universität Darmstadt, Rheinstr. 75, 64295 Darmstadt, Germany. Tel.: +49 6151 869342; fax: +49 6151 869127.

E-mail address: andreas.follner@cased.de (A. Follner).

http://dx.doi.org/10.1016/j.jisa.2016.01.002

2214-2126/© 2016 Elsevier Ltd. All rights reserved.

exist in memory. Recently proposed solutions against such attacks mostly built on CFI (Pappas et al., 2013; Zhang et al., 2013; Zhang and Sekar, 2013) seemed effective, but have been shown to be bypassable (Davi et al., 2014; Göktaş et al., 2014). Section 2 elaborates on these issues in detail.

To battle current exploitation mechanisms we propose ROPocop, a novel tool that mitigates control-flow attacks for x86 Windows binaries using two novel techniques: AntiCRA and DEP+. AntiCRA greatly reduces the risk of successful code-reuse attacks by detecting an unusually high rate of successive indirect branches during the execution of unusually short basic blocks. As different programs can exhibit very different behavior with regard to that aspect, using the same threshold for every program is suboptimal. Therefore, ROPocop comes with a learning mode, which runs ahead of time and determines appropriate thresholds, which can be adopted by the user. However, we do also provide default thresholds which work very well in practice and for a large selection of programs, as our evaluation shows.

Our second contribution, DEP+, implements a variant of a non-executable stack through dynamic binary instrumentation. DEP+ assumes that all code has to reside within a program image, i.e., the .text section of any PE file. This is very similar to DEP (Andersen and Abella, 2004); however, DEP+ cannot be disabled through API calls, thereby eliminating a large class of exploits that are based on such calls. DEP+ enforces all memory to be non-executable, except for the parts to which images are loaded. To this end, DEP+ monitors the loading and unloading of images, checking after each indirect branch whether the program counterpoints outside the known images.

We have implemented ROPocop for Windows x86 using PIN (Luk et al., 2005), a freely-available dynamic program instrumentation framework from Intel. ROPocop requires no access to source code or root privileges, nor debug symbols or changes to the operating system. Measurements using the artificial SPEC CPU2006 suite show an average overhead of 2.4×. More importantly, experiments on real-world applications show only an initially noticeable input lag (caused by the initial dynamic instrumentation) and no stutter. Our evaluation using 11 of the latest code-reuse exploits shows that our tool successfully prevents all code-injection attacks and code-reuse attacks from succeeding, even a highly sophisticated attack that relies solely on code reuse (Li and Szor, 2013). Our envisioned usage of ROPocop is to use it as a last line of defense against exploitation of critical systems, e.g., when a severe vulnerability has been discovered but no patch is available.

To summarize, this work makes the following original contributions:

- AntiCRA, a tunable heuristic detection of code-reuse-attacks like RoP and JoP,
- DEP+, a comparatively fast and very robust implementation of a non-executable stack,
- ROPocop, a dynamic instrumentation tool based on PIN which detects various kinds of control-flow attacks using the above techniques, and
- an empirical evaluation showing that ROPocop ‘s mitigation approach is highly effective and shows tolerable runtime overheads.

We make ROPocop available online as open source, along with all our experimental data (https://sites.google.com/site/ropocopresearch/).

2. Current situation

Exploiting vulnerabilities with the goal to manipulate the program flow was relatively trivial on Windows until the early 2000s, when Microsoft began adapting mitigation techniques. In the simplest cases, an attack widely known as *stack smashing* (One, 1996) could be used. Such an attack would leverage unbounded functions, such as strcpy, to write beyond the allocated memory of a buffer. Attackers could thus overwrite the function's stored return address on the stack with an address that points to injected code, which the program will execute after the next return.

To defend against such code injection attacks, Microsoft implemented *Data Execution Prevention* (DEP) (Microsoft), which makes use of a processor's NX (no execute) bit. DEP marks pages which contain data as non-executable, causing a hardware-level exception if execution from within such a page is attempted. This successfully prevents attacks that attempt to execute injected code.

Nevertheless, attackers can bypass DEP in various ways. At present, the most widely used technique is called return-oriented programming (Shacham, 2007). When utilizing RoP, an attacker does not inject any code but instead uses existing code fragments (*gadgets*), which all end with a return instruction. In other words, instead of injecting code, the attacker injects the addresses of the gadgets he wants to execute. On x86, return works by popping an address off the stack into the register EIP and then jumping to that address. By crafting a stack filled with a sequence of gadget addresses, the attacker can execute sequences of gadgets, with the return instruction at the end of each gadget transferring the program flow to the next gadget. Jump-oriented programming (JoP) (Bletsch et al., 2011; Checkoway et al., 2010; Min et al., 2012) is based on the same basic concept as RoP, but uses jmp instructions to transfer control flow to the next gadget. In the following, we refer to both RoP and JoP attacks as *code-reuse attacks*.

The success of code-reuse attacks depends on the availability of useful gadgets on the target platform and the complexity of the code the attacker wants to run. In practice, however, most systems are vulnerable to such code-reuse attacks. Furthermore, RoP attacks are relatively complex to stage, which is why most attacks of this kind do not resort to pure RoP, but rather implement a two-staged approach. The first stage uses RoP to call a Windows API function like `VirtualProtect` (see below) which marks a certain memory region as executable, effectively bypassing DEP. This is followed by the second stage, running code previously injected into that memory region, which can then be executed as normal. Code-reuse attacks work reliably if the memory layout of an application is highly deterministic because an attacker can hard-code the addresses of gadgets directly into the exploit. To mitigate this, Microsoft introduced randomness in the form of ASLR (Howard et al., 2010). ASLR randomizes the order in which images are loaded

Download English Version:

https://daneshyari.com/en/article/458947

Download Persian Version:

https://daneshyari.com/article/458947

Daneshyari.com