StackIDS - Catching Binary Exploits before they Execute a System Call

Jakob Lell Sebastian Koch Joerg Schneider

Modern processor architectures provide mechanisms to distinguish between memory areas containing executable code and memory areas which can be freely modified by the running programs but will not be executed by the processor. On a system using these mechanisms, an attacker exploiting a software flaw is not able to modify the memory with the executed code nor can he place own code in a data area and redirect the control flow there. However, the control flow does not only depend on the program code but also on data values which have to be modified during the program execution. The function return pointers stored on the stack are one prominent example for such data. Moreover, on current architectures the return pointers are stored together with user data, e.g., local variables of the functions.

Recently developed attack techniques like return into libc or return oriented programming (ROP) exploit this mix of user data and control flow relevant data. During an ROP attack, the attacker first gains access to write arbitrary data on the stack, e.g., using a buffer overflow. Then, he modifies the stack such that the respective return addresses point in the existing code to some useful statements close before a return statement. It has been shown that only few of these useful statement combinations—the so called gadgets—have to be found in the existing code to form a Turing complete execution environment for the attacker's exploit code.

However, exploiting a vulnerability in a program usually implies taking over the underlying system. This can only be achieved using at least one system call to communicate with the operating system. While current architectures still mix control flow data and user data, a good detector inside the operating system is needed to distinguish system calls made during normal operation from system calls made by an attackers exploit code.

While previous work mainly focused on detecting anomalies in the sequence of the system calls or on using predefined policies, we propose a new detection mechanism based on the integrity of the stack. When the operating system receives a system call, it compares the current user stack data with the unmodified loaded program code. If the stack shows a function call hierarchy, which cannot be realized by the normal control flow of the program, the system call is not executed and the exploited program is terminated.

In this paper, we define four conditions to detect malicious manipulations of the stack and other data structures indicating a compromised control flow. We have shown with our new monitoring tool StackIDS how the needed data can be collected and analyzed by the operating system. In experiments with realistic software and exploits, we evaluated the effectiveness and the performance of our prototype.