Return-Oriented Programming

Article *in* IEEE Security and Privacy Magazine · November 2012

DOI: 10.1109/MSP.2012.152

2 authors, including:

Marco Prandini
University of Bologna
48 PUBLICATIONS **228** CITATIONS

SEE PROFILE

Editors: Richard Ford, rford@se.fit.edu | Deb Frincke, debfrincke@gmail.com

Return-Oriented Programming

Marco Prandini and Marco Ramilli | University of Bologna, Italy

Buffer overflow is the most common cause of vulnerability for those programs written in languages that leave the burden of memory management to the programmer. Essentially, a buffer overflow occurs when some input to a program is copied into a buffer without a proper check of its actual size. If the input is larger than the buffer, the data overflows past the buffer boundary, overwriting values stored beyond the buffer's end. The attacker's aim is to corrupt the memory in a way that causes a jump to arbitrary code.

Because most buffer overflows are caused by coding errors rather than architectural design flaws, improved design practices and specific developers' training have helped reduce their incidence and impact. Nevertheless, designers of all layers involved in code preparation and execution—hardware, operating system (OS), compilers, runtime environments—have elected to build in countermeasures. This article illustrates these countermeasures, along with attacker responses to them.

OS-Based Countermeasures

A buffer overflow can happen in two distinct contexts, depending on the exploitable input variable: in the stack (stack overflow) if the variable is declared inside a function or in the heap (heap overflow) if the exploited variable is dynamically allocated.

Here, we focus on stack-based buffer overflows, which are typically more dangerous than overflows on the heap, due to the proximity of the function return address. We can further classify this type of overflow into two categories, depending on where attackers try to find the malicious code to be executed: they inject their own shellcode along with the overflowing data into the stack, or they can try to use code that's already present elsewhere in the system.

Let's look more closely at the first scenario. It's more favorable to attackers, who have good control over the shellcode's placement and contents. During the second stage of this attack, attackers drive the execution flow toward the injected payload to run their own instructions on the target machine. To stop this, the OS can implement a countermeasure called data execution prevention (DEP), which consists of adding a flag to the memory management structures. Set to 0, this flag marks the corresponding memory address range as nonexecutable. Because the stack is never meant to store executable code, the OS can set the flag corresponding to the stack area to 0 right during the memory allocation phase at process initialization. Then, even if attackers succeed at injecting code into the stack, they wouldn't be able to execute the payload because the OS would refuse to let the processor jump and fetch the corresponding instructions from the memory area that holds them.

ROP Basics

Erik Buchanan and his colleagues recently described how to force harmless code to perform malicious actions by using an unconventional way of reading it.[1] This technique was introduced in 2007 by Hovav Shacham and later named return-oriented programming (ROP).[2] Attackers can use ROP to circumvent the OS protection that prevents stack portions from being executed.

ROP takes its name from the RETN assembly instruction, the most important one for code executed via a stack-overflow attack because it's responsible for driving the process control out of the attacked process flow. Indeed, RETN takes (pops) the saved instruction pointer from the stack and upgrades the registers `%esp` and `%eip`. More precisely, it adds 4 bytes to `%esp` and puts the popped value back to `%eip`. If attackers can exploit a buffer overflow on the target system, they can place a value of choice in the memory cell at `%ebp + 4`, which means that, as soon as the execution flow encounters RETN, the instruction pointer is loaded and the execution

 Copublished by the IEEE Computer and Reliability Societies

continues from the pointed memory cell. For this property, each instruction sequence ending in the IA32 instruction "RETN" is called a *gadget*. If attackers know where to find various gadgets, they can start a return chain to execute arbitrary code.

The first gadget in the chain is invoked by placing its address on the stack (`%ebp + 4`) with a buffer-overflow attack; after its execution, the control flow gets back to the attackers because every gadget ends with an RETN. By modifying the next return address on the stack, attackers can redirect the control flow to another gadget until they want to end the code execution.

Because gadget chaining is essentially a sequence of jumps to pieces of existing code, being able to exploit it for malicious purposes might seem impossible. However, the ROP technique uses code misalignment to forge new instructions from code already loaded in memory. This technique is made possible by IA32's so-called *language density*: the IA32 architecture is programmed through a set of instructions so large that almost every sequence of bytes could be interpreted as a valid IA32 instruction.

In his paper on ROP, Shacham proved that misalignment produces a Turing-complete language, giving the programmer the ability to write any possible algorithm. A language is called Turing complete or computationally universal if it simulates any single-taped Turing machine. In the real world, a Turing-complete language can solve a computational problem using a finite amount of resources. Figure 1 shows how to forge new instructions from memory data via data misalignment. In its standard, intended alignment, the machine code represents the instructions `movl %edx,0x4(%eax)` and `movl %eax, 0x0805d0ff`. By applying a misalignment of 2 bytes—that is, by making the processor skip the values 0x89 and 0x50—the processor reads a sequence representing the instructions `addb 0xa3, %al`, and `call *%eax`.

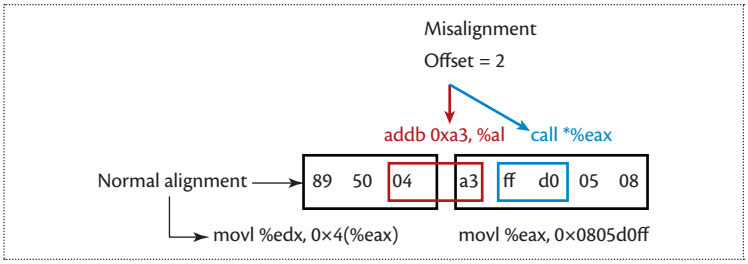

Figure 1. Return-oriented programming misalignment. X86 instructions can be found in the program byte-stream by reading it with a nondefault offset.

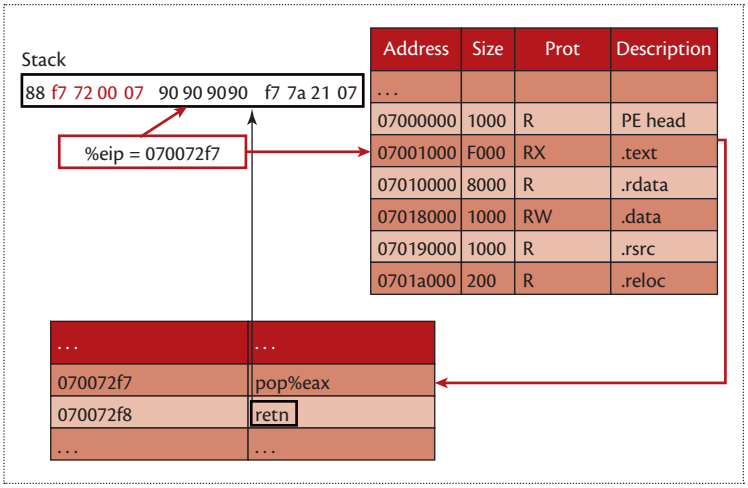

Figure 2. Two gadgets chained together. The addresses of the first and second gadgets are placed on the stack, separated by data consumed by the first gadget, so that the RETN actually causes a jump to the second gadget.

The most important result of "gadget-based programming" for a malicious attacker is that it becomes quite easy to bypass DEP. A chain of gadgets resides in executable memory areas, as opposed to classic injected code that resided in the stack, so DEP countermeasures can't deny its execution. Figure 2 shows a partial example, limited to two steps of a chain. The attacker injects two addresses, `0x070072f7` and `0x07217af7`, into the stack. The first address is placed through a buffer overflow in `%ebp + 4`; the second address is placed 4 bytes ahead. When the processor returns from the exploited function, it returns to `0x070072f7`, which is a valid gadget placed into the `.text` section. The gadget in the example actually pops 4 bytes from the stack, saving them to `%eax`, and then it returns. The saved register holds the next 4 bytes, and the gadget returns later. The return instruction makes the processor fetch the next instruction from the cell pointed by `%esp`, which holds the next gadget address. Because the first gadget as a side effect pops 4 bytes from the stack, the same amount of NOP instructions are inserted in between the two gadgets to achieve proper invocation of the second one.

Typical Implementation

Although attackers could in principle write any program by assembling

complex return chains, ROP has so far been used to make existing buffer-overflow attacks work even in the presence of OS countermeasures. This goal is easily reached by calling system APIs that disable DEP, thus making code on the stack executable again. To this end, the most common implementations of ROP-based attacks on Windows call `SetProcessDEPPolicy()`.

The attacker attains his goal by injecting the following sequence of components on the stack:

- an NOP sequence, which compensates the uncertainty about the jump destination address;
- the entry point address of `SetProcessDEPPolicy()`;
- the value that should be passed as a parameter to `SetProcessDEPPolicy()` in order to disable DEP protection for the current process;
- an NOP sequence to align the stack addresses; and
- the real shellcode.

Through repeated attempts (also known as memory brute-force attack) the attacker succeeds in causing a `JMP` instruction to land within the NOP sequence, which brings the `%eip` to the first gadget (`SetProcessDEPPolicy`). This takes as input the next value in the stack by popping it out and using it to set the DEP policy off for the current process. Finally, it ends (RETN) pointing out to a final gadget: "JUMP to shellcode," which redirects the instruction pointer to the NOP sequence preceding the real shellcode. It can now be executed because DEP protection is no longer active for the current process. Another common technique is to use the `VirtualAlloc()` to dynamically allocate memory on the heap and `MemCpy()` to copy the injected shellcode into the new created memory area. Once the shellcode is placed into the accessible (and executable) memory space, the control flow is hijacked to point to the first shellcode address.

Evolution

ROP is another weapon in the arms race between attackers and defenders. First came the exploits of naive vulnerabilities in coding, then came DEP, the response from OS designers, and now ROP as an ingenious way to circumvent DEP. The race has already gone much further.[3]

The limit of any compiler-based solution is the overoptimistic assumption that every developer uses a "patched" compiler to build applications.

Code Reuse and ASLR

Instead of looking for gadgets, attackers can try to force a sequence of jumps to various kinds of program fragments, this time choosing from (possibly complex) instruction sequences already available in memory locations designated exactly for the purpose of holding executable code; marking them nonexecutable isn't a viable option. Attackers simply execute code they can't choose (for example, library functions) by injecting in the stack the parameters they want to pass to the code.

The countermeasure for this scenario is to make the guess of the correct location very difficult by randomizing memory allocation. For instance, Microsoft introduced Asynchronous Space Layout Randomization (ASLR) in its second Windows XP service pack as an optional setting—and in Windows Vista as the default option—to mitigate stack attacks via the random generation of process address spaces. Before ASLR's introduction, the Windows loader used to assign the address space required by a process with a very predictable mapping to the virtual memory. ASLR counters this by forcing the Windows loader to randomly assign memory space to every loaded process, making buffer-overflow exploiting processes much harder because attackers don't know in what memory position their payload will be addressed. This defense mechanism is also known as misalignment between the Windows PE header and the process memory addresses.

Detection and Evasion Techniques

The most straightforward approach to detecting whether an ROP attack is running would be to trigger an alarm when the system sees an unusually frequent usage of RETN. Against this simple but effective countermeasure, Stephen Checkoway and his colleagues devised an ROP-like technique that doesn't rely on the return statement.[4] Their attack uses certain instruction sequences that behave like a return (for example, `POP %eax; JMP *%eax;`) and that occur with sufficient frequency in large libraries on (x86) Linux and (ARM) Android to allow creation of Turing-complete gadget sets. The authors described the effects on the stack of return instructions as follows:

- Every return statement retrieves the 4-byte value at the top of the stack and sets the instruction pointer (`%eip`) to that value, so that the instruction beginning at that address is executed.
- Every return statement increases the value of the stack pointer (`%esp`) by four, so that the top of the stack is now located at the word above the word assigned to `%eip`.

Then the researchers found the combinations of IA32 statements that would make the previous two changes on the stack but without using explicit return instructions. Because these sequences don't contain the "trademark" instruction of ROP, the described attack shows a serious potential for bypassing detection.

Michalis Polychronakis and Angelos Keromytis proposed a more general ROP payload detection technique that uses speculative code execution.[5] At each branch, their system examines the potential execution of code that already exists in the address space of a targeted process that could be triggered according to the scanned input data and identifies whether it can be classified as the execution of valid ROP code. The main issue with this approach is that it's fairly expensive in terms of performance, and a tool implementing it could generate false positives if not continuously kept up to date.

Compiler-Based Countermeasures

The aforementioned countermeasures are implemented at runtime, but as we noted, the enabling vulnerability for ROP is introduced at coding time. Kaan Onarlioglu and his colleagues proposed G-Free, a compiler-based solution to this conundrum.[6] G-Free can eliminate all unaligned free-branch instructions inside a binary executable and protect the aligned free-branch instructions from attacker misuse. However, the limit of any compiler-based solution is the overoptimistic assumption that every developer uses a "patched" compiler to build applications. Even with a 100 percent adoption rate of a compiler-based solution, bugs introduced by old, widely deployed software will be available to attackers for many years.

So far, attackers have used ROP to bring old malware back to life by circumventing OS-based countermeasures to stack-overflow vulnerabilities. And because the set of ROP gadgets is a Turing-complete language, gadget-chain programming could be used by itself to generate all the needed functions without injecting shellcode on the stack. Together with some effective variants that have emerged to circumvent available countermeasures, ROP enables malware structures that exhibit a very high degree of variability, thus challenging the design of effective detection techniques. OS designers can double their efforts to stay on top of the sophistication of these attacks, but the ball is really in the court of software engineering itself to devise a solution that closes the buffer-overflow vulnerability once and for all. ■

References

1. E. Buchanan et al., "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC," *Proc. 15th ACM Conf. Computer and Communications Security* (CCS 08), ACM, 2008, pp. 27–38.
2. H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *Proc. 14th ACM Conf. Computer and Communications Security* (CCS 07), ACM, 2007, pp. 552–561.
3. J. Jiang et al., "Hypercrop: A Hypervisor-Based Countermeasure for Return Oriented Programming," *Proc. 13th Int'l Conf. Information and Communications Security* (ICICS 11), Springer-Verlag, 2011, pp. 360–373.
4. S. Checkoway et al., "Return-Oriented Programming without Returns," *Proc. 17th ACM Conf. Computer and Communications Security* (CCS 10), ACM, 2010, pp. 559–572.
5. M. Polychronakis and A.D. Keromytis, "ROP Payload Detection Using Speculative Code Execution," *Proc. 6th Int'l Conf. Malicious and Unwanted Software* (Malware 11), IEEE, 2011, pp. 58–65.
6. K. Onarlioglu et al., "G-Free: Defeating Return-Oriented Programming through Gadget-Less Binaries," *Proc. 26th Ann. Computer Security Applications Conf.* (ACSAC 10), ACM, 2010, pp. 49–58.

Marco Prandini is a research associate at the University of Bologna, Italy. His research interests include high-availability systems administration, system security, and electronic voting systems. Contact him at marco.prandini@unibo.it.

Marco Ramilli is a PhD candidate at the University of Bologna, Italy. His research interests include penetration testing, malware analysis, and electronic voting systems. Contact him at marco.ramilli@unibo.it.