Revisiting Return-Oriented Programming: return-to-libc without Function Calls

William Young
Department of Computer Science
The University of Virginia
wty5dn@virginia.edu

Abstract

Return-oriented programming (ROP) encompasses of a series of techniques that allow a return-to-libc attack to be mounted on x86 executable files without calling any functions or injecting any code. Since its introduction in 2007, ROP has proven itself a formidable foe of security experts and system designers. In this work, we examine the history and foundational principles of ROP, the offensive tools designed to create and deploy ROP attacks, and a series of defensive frameworks developed to mitigate against ROP attacks, paying close attention to the state-of-the-art in ROP attacks and defenses.

1 Introduction

Attack prevention techniques that rely on the "No eXecute" (NX) page protection bit are widespread in modern day architectures and operating systems [7]. NX protections prevent attackers from executing injected code in vulnerable running processes. Return-oriented programming (ROP), first classified by H. Shacham in 2007 [1], encompasses a series of binary exploitation techniques that allow return-to-libc attacks to be mounted on x86 executable files without calling any functions or injecting any code, thus circumventing NX protections. At the time of its introduction, ROP rendered useless the widely deployed W$\oplus$X data execution prevention (DEP) defenses against code injection attacks, which prevented the execution of user-loaded shellcode on the heap and stack. Moreover, unlike traditional return-to-libc attacks, which rely on replacing program return addresses with those of existing library functions, ROP grants attackers a Turing-complete

language consisting of short instruction sequences called *gadgets*. Gadgets are small code fragments that already exist within the running process memory of the vulnerable application. To be useful, gadgets must end with an indirect control transfer instruction that passes control to the next gadget in the sequence or to an attacker-defined memory location [2]. Through careful inspection of the GNU libc code—ubiquitous to most C binaries—an attacker can concatenate together arbitrarily many gadgets in order to construct exploits consisting entirely of valid executable code, thus bypassing DEP defenses.

ROP is particularly dangerous with respect to the advantages it confers to attackers, whom are able to *a priori* select gadgets for use in attacks. Though address space layout randomization (ASLR) is designed to prevent premeditated binary exploits, the existence of ASLR-incompatible shared libraries and fixed-entrypoint libraries results in gadget code remaining static across many different installations and platforms [2]. Even worse, many other defenses against code-reuse attacks such as compiler extensions, control-flow integrity checks, and code randomization fail to make their way into the production software that is most vulnerable to ROP attacks, either due to development constraints, cost, confidentiality, or a combination of all three.[1]

Following its classification in 2007, much research has attempted to codify existing and potential ROP attacks and propose defense mechanisms to mitigate against ROP attacks. On the attacker side, Bittau et al. demonstrate that it is possible to write remote stack buffer overflow exploits without possessing a copy of the target binary by conducting a *blind ROP attack* [8]. Jonathan Salwan's open-source *ROPgadget* project [9] shifts ROP exploit generation into script-kiddie territory by automatically searching for potential gadget-derived exploit sequences in raw binaries. More recently, academic work has explored the concept of so-called *jump-oriented programming* (JOP) [5] [10], a variant of ROP that relies on indirect jumps rather than return instructions to achieve the same effect. Carlini and Wagner [14] demonstrate an approach similar to JOP that they coin *Call Oriented Programming* (COP). COP relies on gadgets that end with memory-indirect calls, differentiating it from JOP, which relies on reading register values to transfer control.

[1] ROP defenses may incur runtime overhead and require painstaking analysis of compiled source code in addition to symbolic debugging information, typically *not* included in production binaries.

On the defensive side, comprehensive (and costly) solutions based on Control Flow Integrity (CFI) [11] prevent ROP in general by enforcing the program control flow graph (CFG), recognizing when control transfers to a location outside the CFG, as it does in ROP attacks. Tools such as *ROPdefender* [3] detect sophisticated ROP attacks without requiring specific side information, instead performing return address validation against values stored in a "shadow stack" at runtime (at a considerable loss of efficiency). Compiler modifications are capable of rewriting programs to produce "return-less" binaries, thus thwarting return-address overwrites [12]. Davi, Sadeghi, and Winandy [13] investigate the use of dynamic binary instrumentation frameworks to detect instruction streams with frequent returns. Hiser et al. [4] propose a novel technique called *Instruction Location Randomization* designed to randomize the location of *all* instructions in a program, thus preventing attackers from executing code re-use attacks such as ROP.

2 Background

Return-oriented programming is a generalization of return-into-libc, an attack method that gained popularity following the introduction of a series of buffer-overflow defense mechanisms such as DEP and W⊕X. In return-into-libc attacks, rather than overflowing a buffer with executable code—which DEP prevents from running—the adversary overwrites a return address on the stack with the static address of an attacker-defined function in libc (often *system*). Thus, execution quite literally "returns into libc" in the course of carrying out the attack.

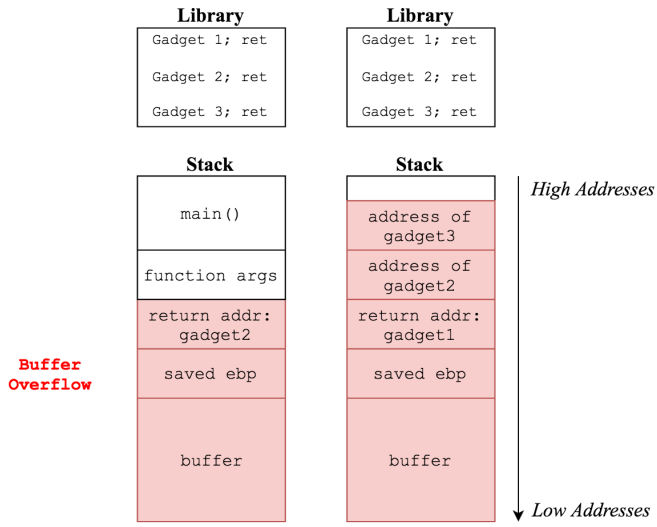

Figure 1: Two simple ROP attacks

ROP generalizes return-into-libc by allowing the attacker to induce arbitrary program behavior through the use of short snippets of library code ending with the ret instruction. The return instruction ensures that each code "gadget" is executed sequentially, and arbitrarily-many gadgets may be executed in sequence to recreate atomic tasks such as read, write, load, store, and many others. In fact, the attack is considered

Turing-complete if the adversary is capable of constructing gadgets for all necessary operations; intuitively, this makes sense, as an attacker could very well construct a series of gadgets that disable DEP and subsequently execute arbitrary code injected in the stack.

Figure 1 demonstrates two ROP attacks. On the left, the function return address is overwritten with the address of gadget 2. Note that the program will likely crash once gadget 2 returns, since it will attempt to return to the value of *function arg*. On the right, a more sophisticated attack is provided—part of the stack frame of *main* is overwritten with additional gadget addresses. When gadget 1 returns, it will trigger gadget 2, and so on, until the program terminates. In order to maintain secrecy, an attacker could complete the ROP "chain" with a legitimate program return address so that the program, rather than crashing, exits gracefully.

3 ROP Attacks

The general goal of a ROP attack is to construct an instruction sequence from gadgets such that the resulting program behavior spawns a shell (often with superuser permissions, though such outcomes depend on the nature of the running program). Acquiring a shell is by far the most effective means of inflicting further damage in an attack, since all system resources are at the attacker's disposal. Before the proliferation of ASLR, executing ROP attacks required attackers to do little more than overflow a buffer and overwrite a return address with that of a gadget (or sequence of gadgets). On contemporary systems, this approach fails. Attackers must both 1) know where the gadgets are located (due to DEP), and 2) determine the location of the executable's text segment in memory (due to ASLR). Unlike 32-bit systems, brute-forcing these addresses on 64-bit systems is nontrivial and time consuming. Thus, all relevant contemporary attacks must somehow circumvent (1) and (2). The remainder of this section will exclusively consider advanced, modern ROP attack methodologies and the tools that make them possible.

A. Blind ROP

In 2014, researchers from Stanford University [8] demonstrated the possibility of launching a remote ROP attack without knowledge of the underlying binary, specifically targeted at server applications that restart following crashes—a technique they coined "blind ROP", or "BROP." A BROP attack is executed in three phases:

1) Stack reading
2) Blind ROP for `write` system call
3) Traditional ROP exploit

In the first phase, the attackers must defeat ASLR. To do so, they develop a new technique that consists of overflowing a single byte of the canary with a value x. If x is correct, the server will not crash. This simple algorithm is repeated up to 256 times for each of the eight bytes of the canary, requiring an average of 128 tries per byte.

Once the canary and saved frame pointer have been compromised, attackers can proceed with the BROP attack. On 64-bit systems, arguments to system calls are passed via the registers, thus an attacker needs gadgets to store a socket, a buffer, the buffer length, and the system call number in `rdi`, `rsi`, `rdx`, and `rax` (respectively). The goal is to find gadgets that manipulate the registers listed above to invoke the `write` system call. This is accomplished by pointing the return address to the text segment and inspecting program behavior. Though the program may crash, the lack of such a crash—combined with other detailed instrumentation behavior—reveals a gadget and its accompanying address. Once the attacker possesses sufficient gadgets to execute the `write` system call, the entire .text segment can be written from memory to the attacker's machine, at which point traditional gadget discovery methods can be employed to build a shellcode (traditional ROP attack).

B. Jump-oriented Programming

Though ROP initially circumvented existing binary defenses at the time of its introduction, its characteristic usage of the stack and consecutive `ret` instructions led to an array of new defenses that detect or prevent those behaviors. In 2010, however, researchers from the North Carolina State University [10] introduced a novel attack termed *jump-oriented programming*, designed to eliminate reliance on the stack and heavy usage of return instructions. Rather than using gadgets ending in `ret`, this attack builds chains consisting of gadgets ending in indirect branches. In order to maintain control of execution (indirect branches carry no such guarantee), attackers must utilize a new class of gadget called a *dispatcher gadget* to govern control flow among the jump-oriented gadgets. The dispatcher gadget maintains a virtual program counter and executes JOP gadgets by advancing through the gadgets one after another, similar to regular program instructions. For this paradigm to function properly, each JOP gadget must terminate with an unconditional jump back into the dispatcher. More formally, the final instruction

in the sequence must load the instruction pointer with the result of a known expression, the value of which must evaluate to the address of the dispatcher by the time the branch is executed.

C. Call-oriented Programming

Similar to jump-oriented programming, call-oriented programming (COP) [14] is an alternate method of executing ROP attacks without using return instructions. Instead of using gadgets ending with returns, COP uses gadgets that end with indirect calls. Unlike in jump-oriented programming, indirect calls in COP transfer control based on values in memory, not those stored in registers. As a result, COP attacks do not require the use of dispatcher gadgets. The attacker, however, must initialize the values of these memory locations *in advance*. Once initialized, the gadget chain executes by pointing to the memory-indirect location containing the address of the next gadget in the sequence. In this work, it was discovered that memory-indirect calls are very common—even more so than ordinary call-preceded returns—in the text segment. This abundance of gadgets unfortunately does not lend itself to simplicity or entirely eliminate the need for ordinary returns—to conduct a COP attack, the attacker must have control of program flow, must overwrite many memory values to enable gadget chaining, and must fully control the stack. Thus, ROP attacks are usually deployed alongside COP attacks as part of a combined exploit.

D. Software Tools to Aid in Gadget Identification

Following the emergence of ROP, a number of open-source tools were released that allow users to identify useful ROP gadget sequences in target binaries. *ROPgadget* [9] is a binary scanner written in Python that searches user-provided binaries to identify useful gadget sequences. *ROPgadget* is currently capable of supporting architectures such as ARM, x86, x64, MIPS, Sparc, and PowerPC. The most recent release (v5.4) adds support for `jmp` and `call`, allowing users to identify JOP and COP exploit gadgets. Similarly, *Ropper* [15] enables users to search for useful gadgets in x86, x64, ARM, MIPS, and PowerPC binaries. Users can select from predefined ROP chain generators that perform activities such as spawning shells and removing memory protections in addition to specifying their own.

4 ROP Defenses

As a direct result of the danger and prevalence of ROP attacks, a series of defensive measures have arisen to detect or prevent them from occurring.

A. Return-address Monitoring

ROPdefender [3] is a tool developed by German researchers that aims to prevent traditional ROP attacks by carefully tracking calls and returns as they are encountered during program execution. Unlike compiler-based tools, *ROPdefender* can be deployed alongside any existing binary, assuming by default there is no access to source code. Before an instruction is executed, *ROPdefender* intercepts the instruction to determine whether it is a call. If so, a copy of the pushed return address is saved on a *shadow stack*. Similarly, if the instruction is a return, it is compared against the address at the top of the shadow stack. If a mismatch is encountered, a ROP attack has occurred and execution terminates. *ROPdefender* is able to account for edge cases in which false positives may occur, such as when 1) a function is called but does not return; 2) control is transferred without the use of a call instruction; and 3) a different return address is computed while the function is running. These three exception-tolerant qualities, combined with reasonably low overhead (approx. 2x, due to instrumentation) make *ROPdefender* a uniquely useful tool in the defense against traditional ROP.

B. Control Flow Integrity

ROP attacks inherently involve altering the program control flow to execute attacker-selected code. As a result, in the event of an attack, program control flow deviates from the predefined Control Flow Graph (CFG). Due to this certainty of effect, the enforcement of Control-Flow Integrity (CFI) can prevent such attacks from occurring. Abadi et al. [11] formally verify the safety properties guaranteed by CFG enforcement and introduce an implementation for the x86 architecture. Their implementation relies on the *Vulcan* binary instrumentation system to construct a CFG of the program being instrumented, requiring neither recompilation nor source code access.

When calls and returns are encountered, the CFI instrumentation compares source and destination addresses. Two destinations are considered equivalent when the CFG contains edges to each from the same set of sources. Unfortunately, CFI enforcement, while provably secure, comes at a great cost in performance—up to 50% in some tested

binaries, with an average 8% increase in binary size (CFG and CFI rewrites) and 16% increase in execution overhead. These qualities generally render CFI-enforcement unusable in practice, though researchers hope that future advances in binary instrumentation will make such approaches more palatable.

C. Instruction Location Randomization

In 2012, Hiser et al. [4] introduced a novel technique called Instruction Location Randomization (ILR) designed to randomize the location of all instructions in a program, thus thwarting ROP attacks. Like *ROPdefender*, ILR can operate on arbitrary binaries and does not require access to source code or recompilation. ILR employs a non-sequential execution model in which successor instructions are explicitly assigned in a *fallthrough map*, entirely independent of instruction location. Thus, instructions may be spread throughout the memory space without fear of attackers predicting an instructions' relative location based on that of another instruction in the sequence. A process-level virtual machine (PVM) fetches and executes instructions according to the mapping prescribed in the fallthrough map. Though ILR incurs an average overhead of 13%, it randomizes 99.96% of the total gadgets reported by ROPgadget v3.1.

D. ROP Without Returns

Li et al. [12] raise concern with so-called *return-oriented rootkits*, a serious class of OS exploits in which "good" kernel code is used to perform malicious actions via ROP-inspired gadget-hunting. To remedy the issue, they propose to alter compiler design such that an OS kernel may be generated that is entirely devoid of return instructions. Since merely replacing return instructions with semantically-equivalent jump instructions would still allow attackers to conduct ROP-inspired attacks, the researchers also propose a technique they call *return indirection*, in which all valid return addresses are stored in a centralized return address table rather than in the stack frame. A "return index" is automatically pushed and later popped off the stack corresponding to an entry in the table. Since each valid return address has to point to an instruction immediately following a call instruction, the table is static and needs to be generated only once prior to usage. The resultant kernel image size increases by 9% and contains 10% more instructions, but eliminates 18,330 return opcodes (100%). Performance runtime is degraded by an average of 5-15% depending on the nature of the task being executed, falling within an acceptable performance threshold given that return-oriented rootkits are completely nullified.

5 Conclusion

Since its emergence only a decade ago, ROP has proven to be a challenging area of security research; this is primarily due to the fact that every new defense mechanism is leapfrogged by a new, previously-undiscovered attack vector. Even though jump-oriented and call-oriented programming circumvented many of the original ROP defenses, contemporary defense measures such as control flow integrity analysis, totally randomized instructions, and return-less assembly show great promise. As researchers continue to devise more and more advanced defense measures, however, the attacks are only likely to grow in complexity and severity.

6 References

[1] H. Schacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security,* 2007.

[2] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-oriented Programming Using In-place Code Randomization," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[3] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[4] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'D My Gadgets Go?" in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[5] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[6] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133375.2133377

[7] M. Miller, T. Burrell, and M. Howard, "Mitigating software vulnerabilities," Jul. 2011, http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788.

[8] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in Proc. 35th IEEE Sym. Security & Privacy (S&P), 2014, pp. 227–242.

[9] J. Salwan, *ROPgadget*, https://github.com/JonathanSalwan/ROPgadget.

[10] T. Bletsch, X. Jiang, V. Freeh, "Jump-Oriented Programming: A New Class of Code-Reuse Attack," in *Proceedings of the 17th Annual ACM Conference on Computer and Communications Security,* 2010.

[11] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In V. Atluri, C. Meadows, and A. Juels, editors, *Proceedings of CCS 2005*, pages 340–53. ACM Press, Nov. 2005.

[12] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In G. Muller, editor, Proceedings of EuroSys 2010, pages 195–208. ACM Press, Apr. 2010.

[13] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In N. Asokan, C. Nita-Rotaru, and J.-P. Seifert, editors, Proceedings of STC 2009, pages 49–54. ACM Press, Nov. 2009.

[14] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In USENIX Security Symposium, 2014.

[15] *Ropper:* gadget-finding tool for ROP exploits. https://github.com/sashs/Ropper