

Return-oriented programming

From Wikipedia, the free encyclopedia

Return-oriented programming (ROP) is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses such as non-executable memory and code signing.^[1]

In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences, called "gadgets".^[2] Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code. Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defenses that thwart simpler attacks.

Contents

- 1 Background
 - 1.1 Return-into-library technique
 - 1.2 Borrowed code chunks
- 2 Attacks
 - 2.1 x86 architecture
- 3 Defenses
- 4 See also
- 5 References
- 6 External links

Background


Return-oriented programming is an advanced version of a stack smashing attack. Generally, these types of attacks arise when an adversary manipulates the call stack by taking advantage of a bug in the program, often a buffer overrun. In a buffer overrun, a function that does not perform proper bounds checking before storing user-provided data into memory will accept more input data than it can store properly. If the data is being written onto the stack, the excess data may overflow the space allocated to the function's variables (e.g., "locals" in the stack diagram to the right) and overwrite the return address. This address will later be used by the function to redirect control flow back to the caller. If it has been overwritten, control flow will be diverted to the location specified by the new return address.

In a standard buffer overrun attack, the attacker would simply write attack code (the "payload") onto the stack and then overwrite the return address with the location of these newly written instructions. Until the late 1990s, major operating systems did not offer any protection against these attacks; Microsoft Windows provided no buffer-overrun protections until 2004.^[3] Eventually, operating systems began to combat the exploitation of buffer overflow bugs by marking the memory where data is written as non-executable, a technique known as data execution prevention. With data execution prevention enabled, the machine would refuse to execute any code located in user-writable areas of memory, preventing the attacker from placing

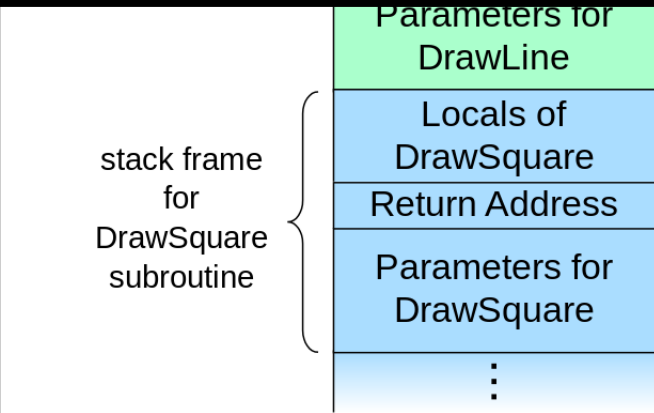
payload on the stack and jumping to it via a return address overwrite. Hardware support for data execution prevention later became available to strengthen this protection.

Return-into-library technique

The widespread implementation of data execution prevention made traditional buffer overflow vulnerabilities difficult or impossible to exploit in the manner described above. Instead, an attacker was restricted to code already in memory marked executable, such as the program code itself and any linked shared libraries. Since shared libraries, such as libc, often contain subroutines for performing system calls and other functionality potentially useful to an attacker, they are the most likely candidates for finding code to assemble an attack.

Pin now, read later! Save any page by clicking  above.

[See how](#)
[Not now](#)



An example layout of a call stack. The subroutine DrawLine has been called by DrawSquare - the stack is growing upwards in this diagram.

In a return-into-library attack, an attacker hijacks program control flow by exploiting a buffer overrun vulnerability, exactly as discussed above. Instead of attempting to write an attack payload onto the stack, the attacker instead chooses an available library function and overwrites the return address with its entry location. Further stack locations are then overwritten, obeying applicable calling conventions, to carefully pass the proper parameters to the function so it performs functionality useful to the attacker. This technique was first presented by Solar Designer in 1997,^[4] and was later extended to unlimited chaining of function calls.^[5]

Borrowed code chunks

The rise of 64-bit x86 processors brought with it a change to the subroutine calling convention that required the first argument to a function to be passed in registers instead of on the stack. This meant that an attacker could no longer set up a library function call with desired arguments just by manipulating the call stack via a buffer overrun exploit. Shared library developers also began to remove or restrict library functions that performed functions particularly useful to an attacker, such as system call wrappers. As a result, return-into-library attacks became much more difficult to successfully mount.

The next evolution came in the form of an attack that used chunks of library functions, instead of entire functions themselves, to exploit buffer overrun vulnerabilities on machines with defenses against simpler attacks.^[6] This technique looks for functions that contain instruction sequences that pop values from the stack into registers. Careful selection of these code sequences allows an attacker to put suitable values into the proper registers to perform a function call under the new calling convention. The rest of the attack proceeds as a return-into-library attack.

Attacks

Return-oriented programming builds on the borrowed code chunks approach and extends it to provide Turing complete functionality to the attacker, including loops and conditional branches.^{[7][8]} Put another way, return-oriented programming provides a fully functional "language" that an attacker can use to make a compromised machine perform any operation desired. Hovav Shacham published the technique in 2007^[9] and demonstrated how all the important programming constructs can be simulated using return-oriented programming against a target application linked with the C standard library and containing an exploitable buffer overrun vulnerability.

A return-oriented programming attack is superior to the other attack types discussed both in expressive power and in resistance to defensive measures. None of the counter-exploitation techniques mentioned above, including removing potentially dangerous functions from shared libraries altogether, are effective against a return-oriented programming attack.

x86 architecture

Although return-oriented programming attacks can be performed on a variety of architectures,^[9] Shacham's paper and a majority of follow-up work focuses on the Intel x86 architecture. The x86 architecture is a variable-length CISC instruction set. Return-oriented programming on the x86 takes advantage of the fact that the instruction set is very "dense", that is, any random sequence of bytes is likely to be interpretable as some valid set of x86 instructions.

It is therefore possible to search for an opcode that alters control flow, most notably the return instruction (0xC3) and then look backwards in the binary for preceding bytes that form possibly useful instructions. These sets of instruction "gadgets" can then be chained by overwriting the return address, via a buffer overrun exploit, with the address of the first instruction of the first gadget. The first address of subsequent gadgets is then written successively onto the stack. At the conclusion of the first gadget, a return instruction will be executed, which will pop the address of the next gadget off the stack and jump to it. At the conclusion of that gadget, the chain continues with the third, and so on. By chaining the small instruction sequences, an attacker is able to produce arbitrary program behavior from pre-existing library code. Shacham asserts that given any sufficiently large quantity of code (including, but not limited to, the C standard library), sufficient gadgets will exist for Turing-complete functionality.^[9]

An automated tool has been developed to help automate the process of locating gadgets and constructing an attack against a binary.^[10] This tool, known as ROPgadget, searches through a binary looking for potentially useful gadgets, and attempts to assemble them into an attack payload that spawns a shell to accept arbitrary commands from the attacker.

Defenses

A number of techniques have been proposed to subvert attacks based on return-oriented programming.^[11] Most rely on randomizing the location of program and library code, so that an attacker cannot accurately predict the location of instructions that might be useful in gadgets and therefore cannot mount a successful return-oriented programming attack chain. One fairly common implementation of this technique, address

space layout randomization (ASLR), loads shared libraries into a different memory location at each program load. Although widely deployed by modern operating systems, ASLR is vulnerable to information leakage attacks and other approaches to determine the address of any known library function in memory. If an attacker can successfully determine the location of one known instruction, the position of all others can be inferred and a return-oriented programming attack can be constructed.

This randomization approach can be taken further by relocating all the instructions of the program separately, instead of just library locations.^[12] This requires extensive runtime support, such as a software dynamic translator, to piece the randomized instructions back together at runtime. This technique is successful at making gadgets difficult to find and utilize, but comes with significant overhead.

Another approach, taken by kBouncer,^[13] modifies the operating system to track that return instructions actually divert control flow back to a location immediately following a call instruction. This prevents gadget chaining, but carries a heavy performance penalty.^[13] In addition, it is possible to effect a return-oriented programming attack without using return instructions at all, but instead via other control-flow-modifying instructions such as jumps. kBouncer is not effective against this type of modified attack.

See also

- Threaded code – return-oriented programming is a rediscovery of threaded code

References

- Shacham, Hovav; Buchanan, Erik; Roemer, Ryan; Savage, Stefan. "Return-Oriented Programming: Exploits Without Code Injection". Retrieved 2009-08-12.
- Buchanan, E.; Roemer, R.; Shacham, H.; Savage, S. (October 2008). "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC". *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08* (PDF). pp. 27–38. doi:10.1145/1455770.1455776. ISBN 978-1-59593-810-7.
- Microsoft Windows XP SP2 Data Execution Prevention (<http://technet.microsoft.com/en-us/library/cc700810.aspx>)
- Solar Designer, *Return-into-lib(c) exploits* (<http://seclists.org/bugtraq/1997/Aug/63>), Bugtraq
- Nergal, Phrack 58 Article 4, *return-into-lib(c) exploits* (<http://phrack.org/issues/58/4.html>)
- Sebastian Krahmer, *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique* (<http://users.suse.com/~krahmer/no-nx.pdf>), September 28, 2005
- Abadi, M. N.; Budiu, M.; Erlingsson, Ú.; Ligatti, J. (November 2005). "Control-Flow Integrity: Principles, Implementations, and Applications". *Proceedings of the 12th ACM conference on Computer and communications security - CCS '05*. pp. 340–353. doi:10.1145/1102120.1102165. ISBN 1-59593-226-7.
- Abadi, M. N.; Budiu, M.; Erlingsson, Ú.; Ligatti, J. (October 2009). "Control-flow integrity principles, implementations, and applications". *ACM Transactions on Information and System Security* **13**: 1. doi:10.1145/1609956.1609960.
- Shacham, H. (October 2007). "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*. pp. 552–561. doi:10.1145/1315245.1315313. ISBN 978-1-59593-703-2.
- Jonathan Salwan and Allan Wirth, *ROPgadget - Gadgets finder and auto-roper* (<http://shell-storm.org/project/ROPgadget/>)
- Skowyra, R.; Casteel, K.; Okhravi, H.; Zeldovich, N.; Streilein, W. (October 2013). "Systematic Analysis of Defenses against Return-Oriented Programming". *Research in Attacks, Intrusions, and Defenses* (PDF). Lecture Notes in Computer Science **8145**. pp. 82–102. doi:10.1007/978-3-642-41284-4_5. ISBN 978-3-642-41283-7.

12. Hiser, J.; Nguyen-Tuong, A.; Co, M.; Hall, M.; Davidson, J. W. (May 2012). "ILR: Where'd My Gadgets Go?". *2012 IEEE Symposium on Security and Privacy*. pp. 571–585. doi:10.1109/SP.2012.39. ISBN 978-1-4673-1244-8.
13. Vasilis Pappas. *kBouncer: Efficient and Transparent ROP Mitigation* (<http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf>). April 2012.

External links

- "Computer Scientists Take Over Electronic Voting Machine With New Programming Technique". Science Daily. Aug 11, 2009.
- "Return-oriented Programming Attack Demo video".
- AntiJOP: a program that removes JOP/ROP vulnerabilities from assembly language code (<http://zsmith.co/antijop.html>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Return-oriented_programming&oldid=717750039"

Categories: Computer security exploits

-
- This page was last modified on 29 April 2016, at 14:03.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.