Return-oriented programming

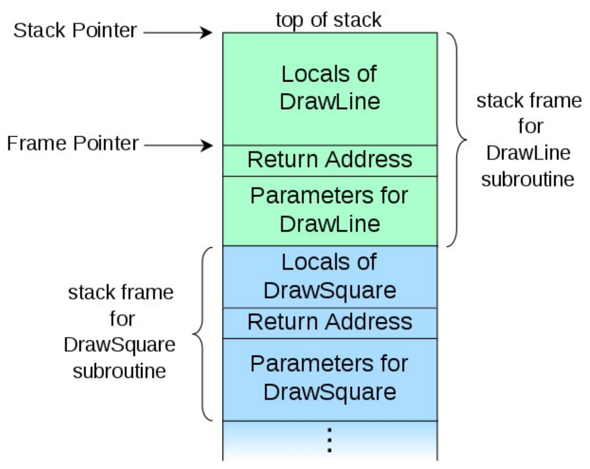
From Wikipedia, the free encyclopedia
Jump to: navigation, search

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 (W xor X technique) 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 that are already presented in machine's memory, 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

Background[edit]



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

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.

With data execution prevention, an adversary cannot execute maliciously injected instructions because a typical buffer overflow overwrites contents in the data section of memory, which is marked as non-executable. To defeat this, a return-oriented programming attack does not inject malicious code, but rather uses instructions that are already present, called "gadgets", by manipulating return addresses. A typical data execution prevention cannot defend against this attack because the adversary did not use malicious code but rather combined "good" instructions by changing return addresses; therefore the code used would not be marked non-executable.

Return-into-library technique[edit]

See also: Return-to-libc attack

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.

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[edit]

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[edit]

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.

on x86 architecture[edit]

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.

on ASLR[edit]

The address space layout randomization also has vulnerabilities. According to the paper of Shacham et al,[11] the ASLR on 32-bit architectures is limited by the number of bits available for address randomization. Only 16 of the 32 address bits are available for randomization, and 16 bits of address randomization can be defeated by brute force attack in minutes. For 64-bit architectures, 40 bits of 64 are available for randomization. In 2016, brute force attack for 40-bits randomization are possible, but it is unlikely to go unnoticed. Also, randomization can be defeated by de-randomization technique.

Also, even with perfect randomization, leakage of memory contents will help to calculate the base address of a DLL at runtime.[12]

No use of return instruction[edit]

According to the paper of Checkoway et al,[13] it is possible to perform return-oriented-programming on X86 and ARM architectures without using return instruction: 0xC3. They instead used carefully crafted instruction sequences that already exist in the machine to behave like return instruction. Return instruction has 2 effects: (1) the instruction searches for the four-byte value at the top of the stack, and set the instruction pointer to the value. (2) it increases the stack pointer value by four. On X86 architecture, sequences of jmp and pop instructions can act as a return instruction. On ARM, sequences of load and branch instructions can act as a return instruction.

Since this new approach does not use return instruction, it has negative implications for defense. When defense program check not only for several returns but also for several jump instructions, this attack will be detected.

Defenses[edit]

Unbroken defense[edit]

The G-free technique was developed by Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. It is a practical solution against any possible form of return-oriented programming. The solution eliminates all unaligned free-branch instructions (instructions like RET or CALL which attackers can use to change control flow) inside a binary executable, and protects the free-branch instructions from being used by an attacker. The way G-free protects the return address is similar to the XOR canary implemented by StackGuard. Also, it checks authenticity of function calls by appending validation block. If the decreed result is not found or does not match, then G-free causes the application to crash.[14]

Address Space Layout Randomization[edit]

A number of techniques have been proposed to subvert attacks based on return-oriented programming.[15] 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 and/or other program state (registers and stack objects) of the program separately, instead of just library locations.[16][17][18] 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, modifies the operating system to verify 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,[*clarification needed*] and is not effective against jump-oriented programming attacks which alter jumps and other control-flow-modifying instructions instead of returns.[19]

DEP/NX[edit]

Data Execution Prevention/No-execute is a defense application developed by Microsoft. DEP/NX alone does not provide perfect protection against all memory-based vulnerabilities, but as with ASLR technologies, it can provide advanced protection on memory based vulnerabilities.

SEHOP[edit]

Structured Exception Handler Overwrite Protection is a feature of Windows which provide a protection against the most common stack overflow attacks, especially against attacks on structured exception handler.

Against control Flow attack[edit]

As small embedded systems are increasing due to the expansion of the Internet Of Things, the need of the protection on the embedded system is also increasing. Using Instruction Based Memory Access Control(IB-MAC) implemented in hardware, it is possible to protect low-cost embedded systems against malicious control flow and stack overflow attacks. The protection can be provided by separating data stack and return stack. However, due to the lack of Memory Management Unit of embedded system, the hardware solution cannot be applied to embedded system.[20]

Against Return-Oriented Rootkits[edit]

One approach for defending return-oriented programming would be to create a compiler-based defense mechanism that eliminates return instructions so that an adversary or return-oriented rootkits cannot make return-oriented gadgets. To eliminate return instructions, a technique called return indirection replaces the return address in the stack frame with an index. Thus, an adversary cannot use return addresses in the stack frame anymore. To prevent blocking legitimate instructions, two techniques called register allocation and peephole optimization can be used. The peephole optimization replaces return opcode to non-return opcode.[21]

See also[edit]

Computer security portal

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

References[edit]

1. ^ *Shacham, Hovav; Buchanan, Erik; Roemer, Ryan; Savage, Stefan. "Return-Oriented Programming: Exploits Without Code Injection". Retrieved 2009-08-12.*
2. ^ *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.*
3. ^ Microsoft Windows XP SP2 Data Execution Prevention
4. ^ Solar Designer, *Return-into-lib(c) exploits*, Bugtraq
5. ^ Nergal, Phrack 58 Article 4, *return-into-lib(c) exploits*
6. ^ Sebastian Krahmer, *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*, September 28, 2005
7. ^ *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.*
8. ^ *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.*
9. ^ [a] [b] [c] *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.*
10. ^ Jonathan Salwan and Allan Wirth, *ROPgadget - Gadgets finder and auto-roper*
11. ^ [Shacham et al., 2004] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In Proceedings of the 11th ACM conference on Computer and Communications Security (CCS), 2004.
12. ^ [Bennett et al., 2013] James Bennett, Yichong Lin, and Thoufique Haq. The Number of the Beast, 2013. http://blog.fireeye.com/research/2013/02/ the-number-of-the-beast.html.
13. ^ CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. 2010. Return-oriented programming without returns. In Proceedings of CCS 2010, A. Keromytis and V. Shmatikov, Eds. ACM Press, 559–72
14. ^ ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. 2010. G-Free: Defeating return-oriented programming through gadget-less binaries. In Proceedings of ACSAC 2010, M. Franz and J. McDermott, Eds. ACM Press, 49–58.
15. ^ *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.*
16. ^ *Venkat, Ashish; Shamasunder, Sriskanda; Shacham, Hovav; Tullsen, Dean M. (2016-01-01). "HIPStR: Heterogeneous-ISA Program State Relocation". Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '16. New York, NY, USA: ACM: 727–741. doi:10.1145/2872362.2872408. ISBN 9781450340915.*
17. ^ *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.*
18. ^ [1], Venkat, Ashish; Arvind Krishnaswamy & Koichi Yamada, "Binary translator driven program state relocation"
19. ^ Vasilis Pappas. *kBouncer: Efficient and Transparent ROP Mitigation*. April 2012.
20. ^ FRANCILLON, A., PERITO, D., AND CASTELLUCCIA, C. 2009. Defending embedded systems against control flow attacks. In Proceedings of SecuCode 2009, S. Lachmund and C. Schaefer, Eds. ACM Press, 19–26.
21. ^ LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. 2010. Defeating return-oriented rootkits with “return-less” kernels. In Proceedings of EuroSys 2010, G. Muller, Ed. ACM Press, 195–208

External links[edit]

- *"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

Retrieved from "https://en.wikipedia.org/w/index.php?title=Return-oriented_programming&oldid=783948327"
Categories:

- Computer security exploits

Hidden categories:

- Wikipedia articles needing clarification from June 2015

Navigation menu

Personal tools

- Not logged in
- Talk
- Contributions
- Create account
- Log in

Namespaces

- Article
- Talk

Variants

Views

- Read
- Edit
- View history

More

Search

Search Wikipedia Go

Navigation

- Main page
- Contents
- Featured content
- Current events
- Random article
- Donate to Wikipedia
- Wikipedia store

Interaction

- Help
- About Wikipedia
- Community portal
- Recent changes
- Contact page

Tools

- What links here
- Related changes
- Upload file
- Special pages
- Permanent link
- Page information
- Wikidata item
- Cite this page

Print/export

- Create a book
- Download as PDF
- Printable version

Languages

- Deutsch
- Français
- 한국어
- Italiano
- Português
- Русский
- 中文

Edit links

- This page was last edited on 5 June 2017, at 16:37.
- 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.

- Privacy policy
- About Wikipedia
- Disclaimers
- Contact Wikipedia
- Developers
- Cookie statement
- Mobile view