SECURITY

The Danger of Unrandomized Code

EDWARD J. SCHWARTZ

Ed is currently working on his PhD in computer security at Carnegie Mellon University. His current research interests include operating system defenses and automatic discovery and exploitation of security bugs.
edmcman@cmu.edu

You might not be aware of it, but your operating system is defending you from software attacks: Microsoft Windows, Mac OS X, and Linux all include operating system (OS)–level defenses. In this article, I focus on two modern defenses: address space layout randomization (ASLR) and data execution prevention (DEP). The beauty of these defenses is that they have little runtime overhead and can provide some protection for all programs—even ones with serious vulnerabilities—without requiring access to the program's source code. This flexibility comes with a tradeoff, however: the deployed defenses are not perfect. In this article, I shed some light on how these defenses work, when they work, and how the presence of unrandomized code can allow attackers to bypass them.

To understand the motivation behind ASLR and DEP, we need to understand the exploits they were designed to protect against. Specifically, when we say *exploit* in this article, we are referring to a control flow hijack exploit. Control flow hijack exploits allow an attacker to take control of a program and force it to execute arbitrary code.

At a high level, all control flow exploits have two components: a computation and a control hijack. The computation specifies what the attacker wants the exploit to do. For example, the computation might create a shell for the attacker or create a back-door account. A typical computation is to spawn a shell by including executable machine code called shellcode.

The control hijack component of an exploit stops the program from executing its intended control flow and directs (hijacks) execution to attacker-selected code instead. In a traditional stack-based buffer overflow vulnerability, the attacker overflows a buffer and overwrites control structures such as function pointers or saved return addresses. As long as the attacker knows the address of his shellcode in memory, the attacker can hijack control of the program by overwriting one of these structures with the shellcode's address. In this article, I will call such exploits that use shellcode and a pointer to the shellcode *traditional exploits*. Such an exploit is illustrated in Figure 1.

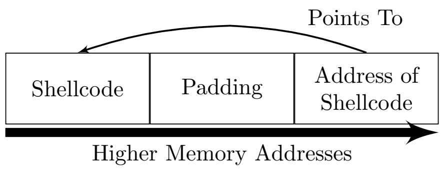

Figure 1: Anatomy of a traditional code injection exploit

Non-Executable Memory

Operating systems have two primary defenses against traditional exploits, called DEP and ASLR. Data execution prevention (DEP) [8] stops an attacker from injecting her own code and then executing it. This effectively prevents the shellcode in traditional exploits from executing successfully. DEP is based on a simple policy: memory should be writable or executable, but never both, at program runtime. To understand the motivation for this policy, consider user input. User input must be written somewhere to memory at runtime, and thus cannot be executable when DEP is enforced. Since shellcode is user input and thus not executable, an exploit utilizing shellcode will crash when DEP is enabled, and the attacker will not be able to execute her computation.

At a high level, DEP sounds like a great defense; it prevents shellcode, and many existing exploits that rely on shellcode will therefore not work. However, one practical limitation is that some programs, such as JIT (just-in-time) compilers, intentionally violate the DEP policy. If DEP is simply enabled for all programs, these programs would stop functioning correctly. As a result, some DEP implementations only protect code modules explicitly marked as DEP-safe.

Another, more fundamental problem is that even when DEP is enabled, *code reuse* attacks can encode a computation in a way that bypasses the defense entirely. The idea is that rather than injecting *new* code into the program, the attacker reuses the executable code that is already there. Because user input is not directly executed, DEP does not prevent the attack.

To perform code reuse attacks, the attacker can find *gadgets*, which are short instruction sequences that perform useful actions. For instance, a gadget might add two registers, or load bytes from memory to a register. The attacker can then chain such gadgets together to perform arbitrary computations. One way of chaining gadgets is to look for instruction sequences that end in `ret`. Recall that `ret` is equivalent to popping the address on the top of the stack and jumping to that address. If the attacker controls the stack, then she can control where the `ret` will jump to. This is best demonstrated with an example.

Assume that the attacker controls the stack and would like to encode a computation that writes `0xdeadbeef` to memory at address `0xcafecafe`. Consider what happens if the attacker finds the gadget `pop %eax; ret` in the executable segment of libc and transfers controls there. The instruction `pop %eax` will pop the current top of the stack—which the attacker controls—and store it in the register %eax. The `ret` instruction will pop the next value from the stack—which the attacker also controls—and jump to the address corresponding to the popped value. Thus, a `ret` instruction allows the attacker to jump to an address of her choosing, and the gadget `pop %eax; ret` allows the attacker to place a value of her choosing in %eax and then jump to an address of her choosing. Similarly, `pop %ebp; ret` allows the attacker to control %ebp and jump somewhere. Finally, if the attacker transfers control to the gadget `movl %eax, (%ebp); ret`, it will move the value in %eax to the memory address specified in %ebp. By executing these three gadgets in that order, the attacker can control the values in %eax and %ebp and then cause the value in %eax to be written to the address in %ebp. In this way the attacker can perform an arbitrary memory write. Figure 2 illustrates a code reuse payload for using these gadgets.

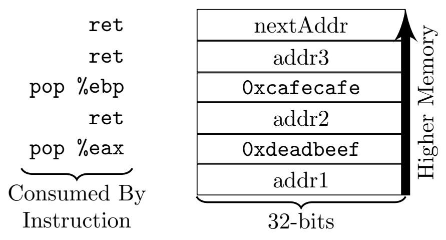

Figure 2: Example of a code reuse payload

Krahmer [3] pioneered the idea of using simple gadgets like these to call functions, which he called the borrowed code chunks technique. Later, Shacham [12] found a set of gadgets in libc that could execute arbitrary programs (or, more formally, arbitrary Turing machines). Shacham's technique is facetiously called return oriented programming (ROP), due to the `ret` at the end of each gadget. Both Krahmer's and Shacham's techniques demonstrate that code reuse can allow very serious attacks when DEP is the only defense enabled.

Randomizing Code Layout

Address space layout randomization (ASLR) [7] is the second primary OS defense. ASLR breaks code reuse attacks by randomizing the location of objects in memory. The idea is that, at some point, exploits need to know something about the layout of memory. For instance, the traditional exploit (shown in Figure 1) uses a pointer to the attacker's shellcode. Similarly, a ROP payload (shown in Figure 2) includes addresses of gadgets in program or library code. If the attacker sets one of these pointers incorrectly, the exploit will fail and the program will probably crash.

It might seem that ASLR completely prevents ROP from succeeding deterministically. If ASLR implementations were perfect, this could be true for some programs. Unfortunately, in practice, ASLR implementations leave at least small amounts of code unrandomized, and this unrandomized code can still be used for ROP.

The details of what gets randomized and when differ for each operating system. On Linux, shared libraries like libc are always randomized, but the program image itself cannot be randomized without incurring a runtime performance penalty. Windows is capable of randomizing both program images and libraries without overhead, but will only do so for modules marked as ASLR-safe when compiled. Although Microsoft's Visual Studio C++ 2010 compiler now marks images as ASLR-safe by default, a great deal of software still contains some modules marked as ASLR-unsafe [9, 6].

These limitations beg the question of why modern operating systems don't randomize all code in memory. On Linux, the reason is because randomizing the program image adds a runtime overhead. Linux programs must be compiled as position-independent executables (PIEs) for the program image to be randomized. On x86 Linux, PIEs run 5–10% slower than non-PIEs [14], but not for x86-64.

Interestingly, the Windows ASLR implementation does not have this problem. The difference is related to how the OSes share code between processes. For instance, if 10 processes of the same program are open, the OS should only have to allocate memory for one copy of the code. One challenge for these shared code mechanisms is that code often needs to refer to objects in memory by their absolute addresses (which might not be known at link time, because of ASLR). Linux uses PIEs to address this problem. PIEs replace each use of an absolute address with a table lookup, which is where the 5–10% overhead comes from. On Windows, the shared code implementation does not require PIEs (and avoids the 5–10% overhead) but typically randomizes each code module only once per boot as a result. Even though Windows can fully randomize program images and libraries with little overhead, it does leave code unrandomized if it is not marked as ASLR-safe, to avoid breaking old third-party programs or libraries which might assume they are always loaded at the same address [6]. This tension between backwards compatibility and security is not just limited to ASLR, either; Windows by default will only protect modules with DEP if they are explicitly marked as safe.

Unrandomized Code

Until fairly recently, it hasn't been clear how dangerous it is to leave small amounts of code unrandomized. Prior work has shown that large unrandomized code bases are very dangerous [12, 3]. For instance, we know we can execute arbitrary programs using libc, which is approximately 1.5 MB. But what can an attacker do with only 20–100 KB of unrandomized code from a program image?

There is already some evidence that even these small amounts of code can lead to successful exploits. Although academic research on ROP has typically focused on Turing completeness, attackers get by in practice with only a few types of gadgets. For instance, rather than encoding their entire computations using ROP, attackers often include shellcode in their exploit but use ROP to disable DEP and transfer control to the shellcode. This can be done by calling the `mprotect` and `VirtualProtect` functions on Linux and Windows, respectively. Many real-world exploits that use these techniques can be found in Metasploit [4], for instance.

Evaluating how applicable these attacks are on large scale is difficult, for two reasons. First, it is intuitively harder to reuse code when there is less code to choose from. This means it will take longer to manually find useful gadgets in the program image of `/bin/true` than it will to find gadgets in libc. Second, the unrandomized code in each program is usually different. This implies that we have to look for different ROP gadgets in every program we would like to exploit. Clearly, constructing ROP attacks by hand for a large number of programs is a tedious and potentially inconsistent task. To evaluate how universal ROP attacks can be, some form of automation is needed. This has motivated some of the joint research with my colleagues, Thanassis Avgerinos and David Brumley.

Specifically, we wanted to know how much unrandomized code is needed for attackers to launch practical attacks such as calling `system('/bin/bash')`. To study this, we built an automated ROP system, Q [11], that is specifically designed to work with small amounts of code, similar to what you might find in a program image. Q takes as input a binary and a target computation and tries to find a sequence of instructions in the binary that is semantically equivalent to the target computation.

The results from our experiments surprised us. Very little code is actually necessary to launch practical ROP attacks. If a function f is linked by the vulnerable program, then Q could create a payload to call f with any arguments in 80% of programs, as long as the program was at least as large as the `/bin/true` command (20 KB). However, attackers often want to call a function g in libc (or another library) that the program did not specifically link. Q was able to create payloads for calling g with any argument in 80% of programs at least as big as `nslookup` (100 KB).

These results are particularly disturbing, because `/bin/true` and `nslookup` are much smaller than the programs often targeted by real attackers.

Other attacks, such as derandomizing libc, can be performed with even greater probability [10]. Derandomizing libc allows the attacker to call functions in libc, but does not necessarily allow the attacker to specify pointer arguments when ASLR randomizes the stack and heap. So, the attacker can call `exit(1)` but not necessarily `system('/bin/bash')`. Even with this restriction, this is a dangerous attack. This type of attack has been shown to be possible in 96% of executables as large as `/bin/true` (20 KB).

Researchers [13] have also noted that the implementations of ASLR on x86 only randomize by 16 bits. This means that an attacker can expect her attack to work after approximately $2^{16} = 65536$ attempts, which is feasible. The suggested fix for this particular problem is to upgrade to a 64-bit architecture.

Defenses

Given that attacks like ROP are so dangerous against ASLR and DEP, it is natural to think of defenses against such attacks. One natural defense against ROP is to disallow unrandomized code in memory. Unfortunately, for performance and backwards compatibility reasons, this is unlikely to happen by default on x86 Windows and Linux. However, Microsoft has released the Enhanced Mitigation Experience Toolkit (EMET) [5], which allows system administrators to force full randomization for selected executables. On Linux, achieving full randomization is possible, but requires recompilation of programs as position-independent executables (PIEs) and incurs a noticeable performance overhead.

A number of other defenses have been proposed to thwart ROP attacks. Many of these defenses are based on the assumption that ROP gadgets always end with a `ret` instruction. Unfortunately, researchers [2] have proven that this assumption is not always true, suggesting that more general defenses, such as control flow integrity (CFI) [1], are needed in practice. Although researchers consider CFI and similar defenses to be low overhead, operating system developers seem unwilling to add defenses with *any* overhead or backwards compatibility problems. Thus, developing a negligible-overhead defense that can prevent ROP without compatibility problems remains an important open problem.

Conclusion

ASLR and DEP are important and useful defenses when used together, but can be undermined when unrandomized code is allowed by the operating system. Recent research [11, 10] has shown that as little as 20 KB of unrandomized code is enough to enable serious attacks when ASLR and DEP are enabled. Unfortunately, modern operating systems currently allow more than 20 KB of unrandomized code, which is unsafe. Until this is remedied, ASLR and DEP are more likely to slow an attacker down than to prevent a reliable exploit from being developed.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti, “Control-Flow Integrity Principles, Implementations, and Applications,” *ACM Transactions on Information and System Security*, vol. 13, no. 1, October 2009.

[2] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy, "Return-Oriented Programming without Returns," *Proceedings of the ACM Conference on Computer and Communications Security*, 2010.

[3] Sebastian Krahmer, "x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique," 2005: http://www.suse.de/~krahmer/no-nx.pdf.

[4] Metasploit: http://metasploit.org.

[5] Microsoft: Enhanced Mitigation Experience Toolkit v2.1, May 2011: z://www.microsoft.com/download/en/details.aspx?id=1677.

[6] Microsoft, SDL Progress Report, 2011: http://www.microsoft.com/download/en/details.aspx?id=14107.

[7] PaX Team, PaX Address Space Layout Randomization (ASLR): http://pax.grsecurity.net/docs/aslr.txt.

[8] PaX Team., PaX Non-executable Stack (NX): http://pax.grsecurity.net/docs/noexec.txt.

[9] Alin Rad Pop (Secunia Research), "DEP/ASLR Implementation Progress in Popular Third-Party Windows Applications": http://secunia.com/gfx/pdf/DEP_ASLR_2010_paper.pdf, 2010.

[10] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi, "Surgically Returning to Randomized lib(c)," *Proceedings of the Annual Computer Security Applications Conference*, 2009, pp. 60–69.

[11] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley, "Q: Exploit Hardening Made Easy," *Proceedings of the USENIX Security Symposium*, 2011.

[12] Hovav Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *Proceedings of the ACM Conference on Computer and Communications Security*, October 2007, pp. 552–561.

[13] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh, "On the Effectiveness of Address-Space Randomization," *Proceedings of the ACM Conference on Computer and Communications Security*, 2004, pp. 298–307.

[14] Ubuntu wiki, Security Features: https://wiki.ubuntu.com/Security/Features?action=recall&rev=52.