Code Execution, Privilege Escalation, and the Memory Underneath

Fri, Sep 25, 6:00 PM EDT · https://scottslab.io/posts/code-execution-privilege-escalation-memory

TL;DR

Arbitrary and remote code execution run with the permissions of whatever process got exploited, which is the entire argument for least privilege and for patching promptly. Privilege escalation turns a normal account into an admin one, often through a buffer overflow, and DEP and ASLR exist specifically to make that harder. Memory leaks, null pointer dereferences, and DLL injection are three different ways bad memory handling turns into a security problem, not just a stability one. And a race condition between a check and the action it authorizes is how the same balance gets spent twice.

Code execution runs as whoever got exploited

Arbitrary code execution and remote code execution both mean the same underlying thing: an attacker gets an unpatched application or operating system to run commands it never should have accepted. The distinction is just position, local versus over the network, and remote is the scarier variant because it needs no foothold to start with.

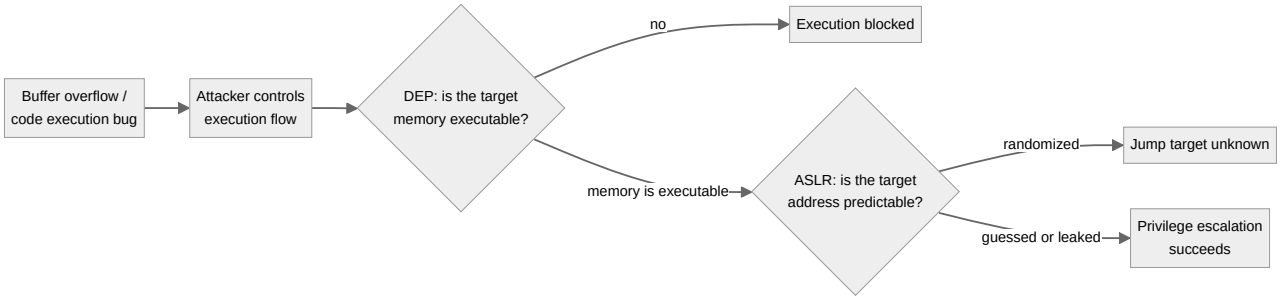
The detail that decides how bad it gets is whose permissions that code runs under. It inherits the privileges of the process it exploited, nothing more and nothing less. A code execution bug in a web server running as a locked down service account is a bad day. The same bug in a process running as root or SYSTEM is the whole box. That is the entire case for least privilege on service accounts: it doesn't stop the exploit, it caps what the exploit is worth. Combine that with patching promptly, since almost every code execution bug that gets used in the wild was fixed by the vendor before it was exploited, and you've addressed the two levers that actually move this risk.

Privilege escalation: getting from user to admin

Privilege escalation is the next step after a foothold: taking a normal account and turning it into an administrative one. A buffer overflow is a classic path there, overwriting memory past the end of an allocated buffer to redirect execution into attacker supplied code running with the target process's privileges, and arbitrary code execution more generally is another.

The mitigations are the same short list every time: validate input so malformed data can't corrupt memory or logic in the first place, keep the OS and applications patched, run services under least privilege accounts so a compromised process has less to escalate into, and lean on the platform's own memory protections.

Two of those protections are worth naming precisely rather than waving at. DEP, data execution prevention, marks memory regions like the stack and heap as non executable, so even if an attacker gets shellcode into memory, the CPU refuses to run it from there. ASLR, address space layout randomization, randomizes where key regions (the executable, libraries, heap, stack) get loaded each time a process starts, so an attacker can't reliably predict the address to jump to. Neither is unbeatable alone, both get chained around in practice, but together they raise the cost of turning a memory bug into working exploit code.



Privilege escalation: getting from user to admin

Three ways memory itself turns into a bug

Memory leaks. Every allocation that never gets freed accumulates until the process runs out of memory and crashes. A reboot clears the symptom, not the cause, so a leaking process on a routine reboot cycle can look healthy indefinitely while it's actually a standing denial of service waiting for the day nobody restarts it on schedule.

Null pointer dereference. The program tries to use a reference that was never set, and at minimum that crashes the app. It can also leak debug information in the resulting error output, stack traces, memory addresses, internal paths, and in the worst case the crash happens partway through a security check, so the control that check was supposed to enforce never runs.

DLL injection. A malicious library gets loaded into a legitimate, already running process's memory space, and from there it executes with that process's identity and hijacks its normal execution path. The malicious code doesn't need its own privileges. It borrows the host process's.

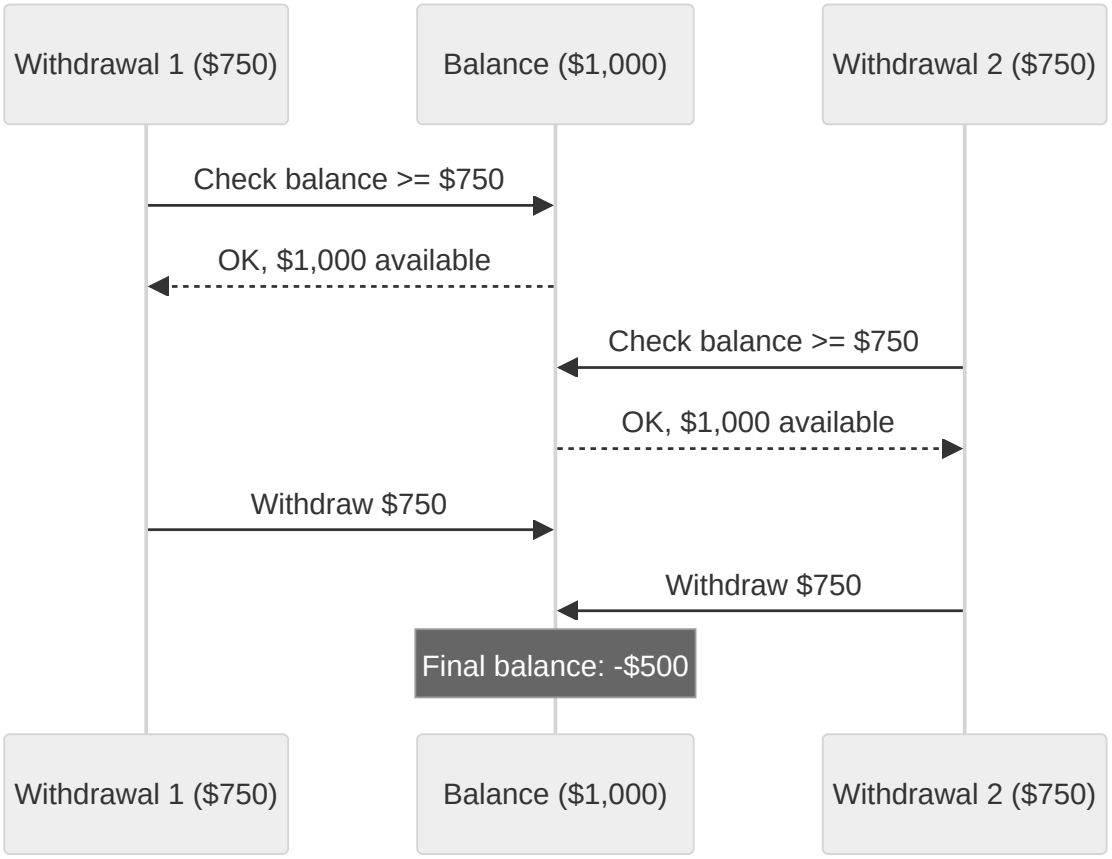
Driver attacks, and why code signing matters

Drivers run at a level of trust most applications never get, which makes them a specific target. Refactoring, in this context, means modifying a driver's source to embed malicious functionality directly, restructuring the code without changing what it outwardly does, so the malware rides along inside something that still passes a functional check. Shimming instead wraps a legitimate driver in a malicious outer layer that intercepts calls, without touching the original driver at all. The defense the platform actually enforces is code signing: the OS refuses to load an unsigned or improperly

signed driver, which forces both techniques to either compromise a legitimate signing key or ship something the OS won't run in the first place.

The race condition: TOCTOU

A time of check to time of use bug is a gap between the moment something gets authorized and the moment it actually happens, and anything can happen in that gap if the resource isn't locked. The classic illustration is an ATM: two withdrawal requests for $750 each hit a $1,000 balance at nearly the same instant. Both check the balance first, both see $1,000, both pass the check, and both proceed, because neither request's check accounted for the other's pending withdrawal. The account ends up $500 overdrawn on a balance that should have blocked the second request outright.



The race condition: TOCTOU

The fix is locking the resource for the duration between check and use, so the second request has to wait for the first to finish before it gets to check anything, rather than checking a snapshot that's already stale. It's a small architectural discipline with a name most developers have heard and a bug most codebases still have somewhere, because it only shows up under real concurrency, and most test suites never generate any.