Secure Coding: Passwords, Errors, and Signed Code

Mon, Sep 28, 6:00 PM EDT · https://scottslab.io/posts/secure-coding-passwords-errors-signed-code

TL;DR

Three defensive-coding habits that get treated as separate topics are really one discipline: never trust what you store, never trust what you're handed, and never trust what you didn't verify. Password storage means salted and slow, hashed with something like Argon2 or bcrypt rather than a fast general-purpose hash, with TLS covering the trip and the session cookie that follows it. Error handling is a security control, not a crash-prevention nicety: the same unvalidated input corrupts memory in unmanaged code and leaks state or skips checks in managed code, and both come from a state diagram missing explicit error states. Code signing is the verification step that turns "we downloaded an update" into "we downloaded the update the vendor actually shipped."

Password storage: hash, salt, and why both matter

Storing a password in plaintext is the fastest way to turn one breach into every account on every other service your users reused that password on. The baseline fix is hashing: run the password through a one-way function and store the output. No inverse operation recovers the original from the hash, only forward computation, so a database dump alone doesn't hand over usable credentials.

Hashing alone still has a hole, and it's the reason rainbow tables exist: a precomputed set of hash outputs for a huge list of candidate passwords, built once and reused against any stolen hash database. If two users both picked `password123`, they get the identical hash, and an attacker who's precomputed that hash once cracks both accounts instantly, along with every other account using the same weak password. Salting closes that hole: a random value, unique per user, gets mixed in before hashing, so the same password produces a different hash for every account. The attacker's precomputed table, built for unsalted inputs, is now useless, since they'd have to recompute it fresh for every distinct salt.

Salting alone still leaves the choice of hash function open, and that choice matters. A fast, general-purpose hash like SHA-256 is built for speed, the wrong property here: billions of guesses a second on commodity GPU hardware, and a salted-but-fast hash still falls to brute force. Password hashing needs the opposite, expense per guess, which is why the algorithms built for the job (Argon2, bcrypt, scrypt) are deliberately slow and, for Argon2 and scrypt, memory-hard: tunable to cost real time and memory per computation, not just the first. Argon2, whose id variant OWASP now recommends by default, won the 2015 Password Hashing Competition; bcrypt and scrypt remain

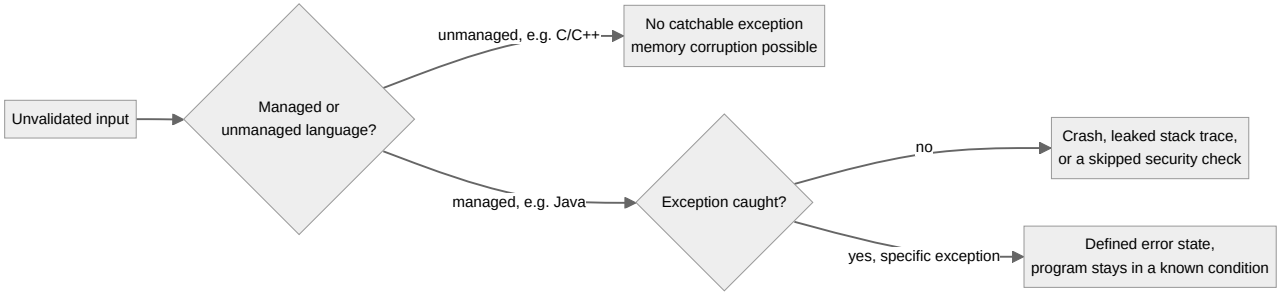
widely deployed. The overhead that feels like a cost at login, milliseconds instead of microseconds, is the point: an attacker at scale pays that cost on every guess too, turning brute force from trivial into impractical.

That's storage. Passwords also have to survive the trip from browser to server, and that's what TLS is for: HTTPS keeps a network observer from reading credentials off the wire during login. It's easy to stop there, but the same connection also protects the session cookie issued after login succeeds. A cookie sent over plaintext HTTP is just as stealable as a plaintext password, and once stolen the attacker doesn't need the password, they just replay the cookie. Transport encryption has to cover the whole session, not just the login itself.

Error handling is a security control

Unhandled input doesn't just produce an ugly error message, it produces an unpredictable program state, and unpredictable states are where security bugs live. What "unpredictable" means depends on the language. In unmanaged C or C++, feeding a routine input it never checked, a length value used straight from user input to size a copy, can corrupt memory directly: there's no language-level bounds check to stop it, and that's the real mechanism behind classic buffer overflow exploitation. In a managed, memory-safe language like Java, that failure doesn't exist: an out-of-bounds access throws a catchable exception before any memory is touched incorrectly, so there's no buffer overflow for a try/catch to prevent there.

The risk in Java is different, not absent. Feed a numeric field the string "apple" and an uncaught `NumberFormatException` can crash the request thread, leak a stack trace carrying internal class names and paths into an error page, or throw mid-way through an unfinished security check, leaving that check never run. An attacker who can reliably trigger that path has found a lever, just not the one a C buffer hands them.



Error handling is a security control

The design mistake underneath both columns of that diagram is architectural, not a missing bounds check or catch. A state diagram for a system is usually drawn as the happy path: state A transitions to state B transitions to state C. If that diagram has no explicit error states, error handling gets bolted on wherever a developer later notices a gap, instead of being part of the design from the start. A state machine that only models success has already decided that failure is undefined behavior,

whether that becomes a corrupted buffer or an exception nobody planned for, and undefined behavior is where an attacker looks first.

Code signing: verifying what you didn't write yourself

Code signing answers a different question: not "did we store this safely" but "is this actually what the vendor sent." The developer generates a key pair and gets the public key certified by a trusted certificate authority, establishing a chain of trust back to a root the OS already recognizes. When the code ships, the developer signs a hash of it with their private key, and that signature travels with the download.

On the receiving end, the OS does two things. It decrypts the signature with the developer's public key, recovering the hash originally signed, then computes the hash of the code that actually arrived. Matching hashes mean the code is authentic (from the holder of that private key) and unmodified (nothing changed after signing). A mismatch, or a certificate that doesn't chain to a trusted root, surfaces a warning instead of silently running unverified code.

I've covered two adjacent angles on this elsewhere: the same check stopping a malicious or shimmed driver from loading, and a plugin ecosystem with no signing, no review, and no sandbox turning a viral AI assistant into a supply-chain incident in under eight hours. Both are this mechanism applied at the point of consequence rather than design: skipping hash-and-signature verification isn't missing a formality, it's accepting arbitrary code on the strength of a filename.

Written by Scott S. — Contact: scott@scottschlangen.com — scottslab.io — https://scottslab.io/posts/secure-coding-passwords-errors-signed-code