**Short answer up-front**

*While you are still* ***tuning*** *your Sysmon rule-set, comment out the bits you don’t want; once you are satisfied with the signal-to-noise ratio, commit a “production” copy in which the unwanted rules are deleted completely.*  
That workflow gives you ⇢ easy rollback during tuning **and** a lean, easy-to-read file for day-to-day use.

Below is the reasoning and some broader hardening tips.

**1. What actually happens when you “comment” a Sysmon rule?**

* Sysmon’s configuration is plain XML. Anything between <!-- … --> is ignored by the parser, so commented rules generate **zero run-time overhead**—they neither slow the driver nor create events. ([whatsupgold.com](https://www.whatsupgold.com/blog/how-to-tune-windows-system-monitor-sysmon))
* The only downside is **human**: the file gets longer and future reviewers may wonder whether the commented block is obsolete or accidentally disabled.

**2. Why keep the rule (commented) during tuning?**

1. **Rapid iteration** – You can re-enable a rule with a single keystroke instead of rewriting XPath conditions from memory.
2. **Audit trail** – Comments provide context (“disabled because it floods 20 k EPS on domain controllers”) which is invaluable six months later.
3. **Safer CI/CD** – When Sysmon restarts it will refuse an invalid XML. Leaving the skeleton of a known-good rule in place makes syntax errors less likely.

The popular *SwiftOnSecurity* template is deliberately “over-commented” for exactly these reasons — it doubles as live config *and* documentation. ([GitHub](https://github.com/SwiftOnSecurity/sysmon-config))

**3. When to delete for good**

| **Stage** | **Recommended action** | **Rationale** |
| --- | --- | --- |
| **PoC / lab** | **Comment** | Fast switching while you learn what is noisy in *your* estate. |
| **Pre-prod** | **Branch your repo** (e.g., prod vs lab) | One branch keeps the historic rules, the other is slim. |
| **Production** | **Delete** anything that is still disabled | Shaves a few KB off the file, avoids reviewer confusion, and makes code reviews shorter. |

A two-file approach is common in mature programs:

* sysmon\_full.xml — everything, mostly commented.
* sysmon\_prod.xml — only the rules that actually fire.  
  Tools like **sysmon-modular** generate both automatically from modules, so you never copy-paste by hand. ([GitHub](https://github.com/olafhartong/sysmon-modular?utm_source=chatgpt.com))

**4. Additional hardening best-practices (2025 edition)**

| **Area** | **Practical tip** |
| --- | --- |
| **Start from a maintained baseline** | Fork *SwiftOnSecurity* or *sysmon-modular* instead of writing from scratch; both are updated for new event IDs (e.g., 34–35 in Sysmon v15). ([GitHub](https://github.com/SwiftOnSecurity/sysmon-config), [GitHub](https://github.com/olafhartong/sysmon-modular?utm_source=chatgpt.com)) |
| **Trim noisy events** | Disable or strictly exclude FileCreateTime, Registry\* and ProcessAccess unless you really need them. |
| **Use include + exclude filters** | Keep onmatch="exclude" rules at the top to drop known chatty paths (AV updaters, browser caches) before they hit your SIEM. |
| **Match schema to binary** | sysmon.exe -s prints the schema; set <Sysmon schemaversion="X.Y"> accordingly or the service will refuse your XML. |
| **Version control** | Store configs in Git; pull requests make every change peer-reviewable and give you free diff/rollback. |
| **Change procedure** | Test with sysmon.exe -c new.xml on a single endpoint, watch event volume for 24 h, then promote. |
| **Measure impact** | Track Events-Per-Second and storage growth; aim < 2k EPS/server unless your SIEM budget is huge. |
| **Centralise logs** | Forward Microsoft-Windows-Sysmon/Operational to your SIEM or Windows Event Forwarding collector; otherwise you lose visibility the moment a box is re-imaged. ([Blumira](https://www.blumira.com/blog/getting-started-with-sysmon?utm_source=chatgpt.com)) |

**5. Recommended workflow checklist**

1. **Clone a baseline** (git clone …sysmon-modular)
2. **Generate** a “full” XML.
3. Deploy to *lab*, run attack-simulation (Atomic Red Team, etc.).
4. **Comment out** high-volume but low-value rules; redeploy.
5. Repeat until event volume is acceptable.
6. Commit the final **production branch** with commented sections hard-deleted.
7. Enforce code-review for every future change.

**Bottom line**

**Comment first, delete later.**  
Keep the experimental notes in a version-controlled “lab” file, but ship a minimal, easy-to-audit configuration to production.

That pattern gives you both agility and long-term clarity—exactly what a hardened logging stack needs.