fuzzing-obstacles

fuzzing-obstacles

> *Catalogue généré le 2026-05-11*

En une phrase

Recueil de techniques pour contourner les "blocages" qui empêchent un fuzzer d'avancer (checksums, horloges aléatoires, validations trop strictes) — en patchant temporairement le code pour qu'il devienne "fuzzable".

Quand l'utiliser

- Quand ton fuzzer stagne et ne trouve plus rien (la couverture ne grimpe plus).
- Quand ton code vérifie un checksum (CRC, SHA) et rejette tout en entrée.
- Quand ton code utilise `time()` ou `random()` (résultat différent à chaque run = fuzzer perdu).
- Quand une validation très stricte bloque toutes les entrées du fuzzer.
- Quand ton code fait des requêtes réseau ou écrit dans une vraie base de données.

Comment l'invoquer

- **Slash command** : `/fuzzing-obstacles`
- **Phrases déclencheurs (texte)** : "fuzzing blocker", "checksum bypass for fuzzing", "anti-fuzzing patterns"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Plein de programmes contiennent des patterns "anti-fuzzing" sans le faire exprès. Par exemple, un format de fichier vérifie son checksum avant traitement : le fuzzer doit deviner la bonne valeur du checksum pour chaque entrée mutée — astronomiquement improbable. Résultat : 99,99 % des entrées sont rejetées immédiatement et le fuzzer n'explore jamais le vrai code.

Autres obstacles classiques : du code qui dépend de l'horloge système (`time()` renvoie une valeur différente à chaque exécution, donc le fuzzer perd le sens du "même input = même résultat"), des générateurs aléatoires non-déterministes, des appels réseau, ou des validations cryptographiques très strictes. Tous ces patterns "cassent" la boucle de feedback du fuzzer.

La solution proposée par cette skill : la **compilation conditionnelle**. Tu ajoutes des `#ifdef FUZZING` autour des morceaux problématiques pour les désactiver uniquement quand on compile pour fuzzer. En production, le code reste 100 % normal. En fuzzing, les checksums sont court-circuités, l'horloge renvoie une valeur fixe, le random est seedé de façon déterministe. D'un coup, le fuzzer voit du progrès et trouve des bugs. C'est une technique reconnue, indispensable pour fuzzer du code "réel".

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `fuzzing-obstacles`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/fuzzing-obstacles/SKILL.md`

Revision #2
Created 2026-05-11 21:19:56 UTC by thymon
Updated 2026-05-11 21:37:29 UTC by thymon