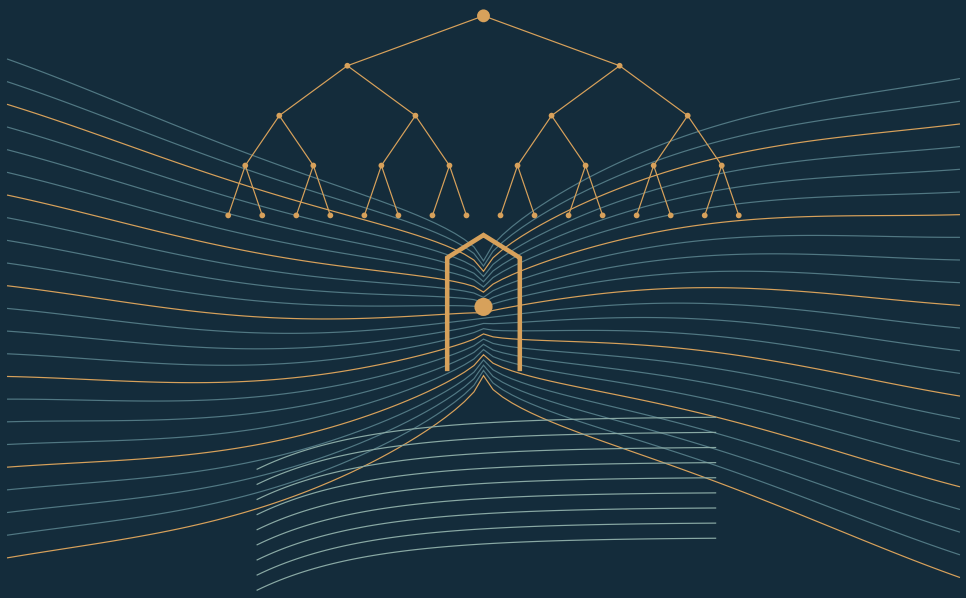
기술 안내서

SIBUNA

웹 방어의 수학과 공학



작업 · 신뢰 · 제한된 상태

Vikrant Rathore 지음, Ronak Rathore · Kanak Rathore 도움

버전 1.0.0-beta.3 · Nov 13, 2026

목차

머리말 **6**

이 책을 읽는 방법 **8**

세 가지 읽기 경로 8

표기 규칙 9

제I부: 요청의 비용 **10**

요청 허용은 경제적 결정입니다 10

작업 증명이 입증하는 것 10

확률변수로 보는 해시 탐색 11

세션은 작업 비용을 분할 상환합니다 12

제한된 상태는 두 번째 예산입니다 12

앞으로 지켜 갈 불변식 12

제II부: 선행 연구와 설계 공간 **14**

Sibuna의 뿌리 14

하나의 바이너리에 담은 세 가지 아이디어 16

기능 비교 17

제III부: 암호학 연구의 기초 **22**

Hashcash: 첫 번째 계층의 수학 22

순차 작업 증명: 두 번째 계층 23

대칭 인증: 챌린지와 세션 26

제IV부: Sibuna 프로토콜 **29**

챌린지 왕복 29

상태 없는 챌린지 레코드 31

검증 순서와 일회 사용 32

세션 쿠키 33

제V부: Rust 무할당 엔지니어링 **36**

연결 루프 36

할당 없는 동시성 상태 47

제VI부: 오토마톤, 트라이, 의미 기반 방화벽 **56**

한 번의 순회로 여러 패턴 매칭하기 56

IPv4와 IPv6를 위한 기수 트라이 58

의미 기반 방화벽 58

선언형 정책 엔진 60

제VII부: 브라우저 작업 증명 엔진 **62**

소스 하나, 타깃 두 개 62

워커 프로토콜 65

보정 66

중간 페이지 67

제VIII부: 실측 평가 **68**
방법론 68
기본 연산 지연 72
전체 제품 비교 75
같은 제공 부하에서의 지연 77
Sibuna와 Anubis 79
콘솔 오버헤드 83
측정하지 않은 것 84
제IX부: 운영과 배포 **86**
운영 구성과 설정 86
콘솔 미리보기 89
배포 토폴로지 106
선언형 정책 112
영속 저장소와 클러스터 113
패키징 118
제X부: 참고 자료와 진단 **122**
엔드포인트 122
지표 123
상태 코드 124
오류 목록 125
정책 스키마 127
저장소 테이블 128
빌드 대상 128
제XI부: 빠른 참조 카드 **130**
빌드 130
실행 130
콘솔(선택형 미리 보기) 130
먼저 알아야 할 플래그 131
/__sibuna/ 아래의 엔드포인트 131
상태 코드 132
헤더 132
정책 파일 골격 133
SQL로 보는 저장소(Edge) 133
진단 134
기억할 숫자 134
인그레스 라우팅 134
일부 연습 문제 해답 **136**
제I부: 비용과 확률 136
제II부: 선행 연구 136
제III부: 암호학 136
제IV부: 프로토콜 137
제V부: 상태와 순서 137

제VI부: 알고리즘 . . . 137
제VII부: 브라우저 엔진 . . . 137
제VIII부: 평가 . . . 138
제IX부: 운영 . . . 138
제X부: 참고 자료 . . . 138
조사 이어 가기 . . . 138
용어집 . . . **139**
참고 문헌 . . . **142**
색인 . . . **143**

© 2026, Vikrant Rathore와 Sibuna 기여자

웹 방어의 수학과 공학.

머리말

요청은 메서드, 경로, 헤더 몇 개를 싣고 서버의 관심을 요구하며 도착합니다. 이 바이트 가운데 어느 것도 그 요청을 처리할 가치가 있다는 근거가 되지 않습니다. Sibuna는 요청과 원본 서버 사이에 결정 하나를 둡니다. 요청을 허용하거나, 거부하거나, 보낸 쪽에 검증 가능한 작업을 요구합니다. 이 책은 이 결정을 수학적 모델에서 메모리 속 바이트까지 따라갑니다.

시스템에는 두 가지 운영 구성이 있습니다. *Gate*는 작업 증명 기반 요청 허용과 노드 단위의 폭주 트래픽 제어를 제공합니다. *Shield*는 여기에 애플리케이션 검사와 정책 결정을 더합니다. 둘 다 정책, 평판, 인시던트 기록을 영속 저장소에 둘 수 있으며, 이 저장소는 노드 하나에 둘 수도 있고 클러스터가 공유할 수도 있습니다. 이런 배포를 *Edge*라고 부릅니다. 유효한 세션은 요청을 허용할 근거가 될 뿐, 요청을 검사에서 면제하지는 않습니다. 영속 상태를 클러스터 전체가 공유하면 상태가 여러 노드에 퍼질 뿐, 로컬 속도 카운터가 전역 할당량이 되지는 않습니다.

엔지니어링의 핵심 질문은 해시가 빠른가가 아닙니다. 신뢰할 수 없는 클라이언트가 입력을 고를 때에도 전체 경로의 비용에 상한이 있는가입니다. 파서는 몇 바이트까지 검사할 수 있을까요? 챌린지 하나가 메모리를 얼마나 차지할 수 있을까요? 클라이언트 하나가 연결을 몇 개나 열어 둘 수 있을까요? 어떤 쓰기를 재시도해도 될까요? 노드나 데이터베이스가 사라진 뒤에도 무엇을 쓸 수 있을까요? 각 답은 불변식과 그 경계를 함께 밝혀야 합니다.

수학도 같은 목적에 쓰입니다. 기대 비용은 지연 시간 보장이 아닙니다. 고정된 커밋먼트에 대한 표본 추출 논증은 모든 적응적 증명자에 대한 증명이 아닙니다. 큐는 순간적인 몰림을 흡수하지만, 처리 속도가 계속 더 느린 소비자를 메워 줄 수는 없습니다. 우리는 작은 모델을 유도하고, 예제를 손으로 풀어 본 뒤, 구현이 모델과 어디서 달라지는지 묻습니다.

이 책은 세 가지 방식으로 쓸 수 있습니다. 순서대로 읽으면 제I부부터 제VII부까지가 하나의 강의 과정입니다. 각 장은 장을 마친 뒤 할 수 있어야 할 일로 시작하고, 예제를 손으로 풀어 본 다음, 연습 문제와 자기 말로 설명해 볼 불변식으로 끝납니다. 일부 해답은 책 끝에 있습니다. 제VIII부부터 제X부까지는 참고 자료입니다. 출처를 밝힌 측정 결과, 모든 플래그와 토폴로지, 모든 엔드포인트, 상태, 오류, 테이블을 담습니다. 제XI부는 터미널 옆에 두고 보는 두 쪽짜리 카드입니다. 책은 용어집과 참고 문헌으로 끝납니다.

제II부는 Sibuna를 Anubis, SafeLine, Cloudflare WAF와 기능별로 나란히 비교합니다. 제VIII부는 기본 연산 하나에서 전체 제품 실행까지 Sibuna의 측정 결과를 제시하고, 같은 부하 생성기 아래에서 Sibuna와 Anubis를 전체 제품으로 비교합니다. 두 부분 모두 독자가 주장을 직접 확인할 수 있게 썼습니다. 비교는 출처를 밝히고, 모든 측정값은 그 값을 옮겨 온 커밋된 결과 파일을 밝힙니다.

독자는 기초 프로그래밍, 로그, 조건부 확률을 알고 있어야 합니다. Rust는 언어 전반을 훑는 대신 소유권과 데이터 배치를 통해 소개합니다. 각 장은 순서대로 읽을 수 있습니다. 비용 모델에서 프로토콜이 나오고, 프로토콜이 상태를 정하며, 상태가 동시성과 저장소 설계를 정합니다. 뒤의 장들은 이 불변식을 측정과 운영 절차로 옮깁니다.

이 책은 구현을 다루는 책이지, 계산 퍼즐이 자동화된 트래픽을 없앤다고 주장하는 책이 아닙니다. 클라이언트는 연산 자원을 살 수 있고, 여러 클라이언트가 주소 하나를 함께 쓸 수 있으며, 애플리케이션 구문은 상한이 있는 탐지기가 다룰 수 있는 범위보다 풍부합니다. 우리가 얻는 쓸

모 있는 결과는 비용과 한계를 구호 뒤에 숨기지 않고 살펴보고, 시험하고, 바꿀 수 있는 시스템입니다.

***Sibuna 1.0.0-beta.3*판, *2026*년 *11*월**
소스, 연습 문제, 재현 가능한 측정 결과는 Sibuna 저장소에 함께 있습니다.

이 책을 읽는 방법

우리는 질문 하나에서 시작합니다. 서버가 지금 내릴 수 있는 가장 값싸고 안전한 결정은 무엇일까요? 프로토콜과 구현을 다루는 장 내내 이 질문을 곁에 둡니다. 잘못된 요청을 허용하는 값싼 연산은 결함입니다. 비용에 상한이 없는 올바른 연산은 또 다른 종류의 결함입니다.

질문	답이 전개되는 곳
퍼즐로 무엇을 얻을까요?	제I부부터 제III부까지: 비용, 확률, 증명의 한계.
Sibuna는 다른 제품과 비교해 어떨까요?	제II부: 계보와 기능 표. 제VIII부: 측정.
세션은 무엇을 뜻할까요?	제IV부: 바인딩, 검증 순서, 만료, 재전송.
메모리는 어디에 쓰일까요?	제V부와 제VI부: 버퍼, 스레드, 테이블, 오토마톤, 게시.
브라우저는 퍼즐을 어떻게 풀까요?	제VII부: 워커 메시지, 메모리, 보정.
제대로 동작하는지 어떻게 알까요?	제VIII부: 하니스 세 개와 각 하니스가 측정에서 빼는 것.
어떻게 실행할까요?	제IX부: 플래그, 토폴로지, 정책, 저장소, 클러스터 구성, 패키징.
방금 그 오류는 무엇이었을까요?	제X부: 엔드포인트, 상태, 오류, 스키마. 제XI부: 카드.

세 가지 읽기 경로

- 설계를 배우려면. 제I부부터 제IV부까지 순서대로 읽고, 제VI부의 풀이 추적을 본 뒤 제VII부를 읽습니다. 연습 문제를 먼저 풀고, 책 끝의 해답과는 그다음에만 비교합니다. 추적이 공유 테이블이나 빌린 슬라이스에 이르면 제V부로 돌아갑니다.
- 배포를 운영하려면. 제IX부와 제XI부를 읽고, 제X부는 찾아볼 수 있게 펼쳐 둡니다. 운영 구성을 고르기 전에 제II부의 "기능 비교"를 읽고, 워커와 연결 한도를 정하기 전에 제VIII부의 "전체 제품 비교"를 읽습니다.
- 코드를 바꾸려면. 제V부와 제VI부, 각 장의 소스 위치, 그리고 그 위치가 인용하는 SID 기록을 봅니다. 제VIII부는 우리가 건드린 부분의 회귀를 어느 하니스가 잡아내는지 설명합니다. 제II부는 변경이 지켜야 할 제약을 나열합니다.

표기 규칙

풀이 예제는 답만이 아니라 중간 상태를 보여 줍니다. 연습 문제는 가정 하나를 바꿔 보게 합니다. 힌트는 첫걸음을 제안하고, 책 끝의 일부 해답으로 추론을 확인할 수 있습니다. 그림은 요청 경로와 백그라운드 작업을 구별합니다. 소스 위치는 설명과 코드가 어긋나 보일 때 살펴볼 구현을 가리킵니다. 제목이 "구현"인 상자는 함수와 그 파일을 밝힙니다. 제목이 "불변식 설명하기"인 상자는 그 개념을 우리가 다시 가르쳐 보게 합니다.

수식에서 b는 비트 단위의 Hashcash 난이도, n은 순차 작업 깊이, t는 개방 수, p는 시행당 성공 확률, K는 시행 횟수, T는 속도 제한기의 방출 간격, τ는 그 버스트 허용치, Q는 큐 용량입니다. 따로 밝히지 않으면 기호는 그 절 안에서만 쓰입니다. 제VIII부의 나노초와 초당 요청 수는 이름이 밝혀진 실행에서 얻은 측정값입니다. 풀이 예제의 숫자는 골라 둔 입력이며, 벤치마크 결과로 내세우는 값이 아닙니다. 통계 모델은 결론을 내리기 전에 가정을 밝힙니다.

제I부

요청의 비용

알고리즘을 고르기 전에 그 알고리즘이 보호하려는 자원을 먼저 밝힙니다. 이 장에서는 이후의 모든 설계 결정을 재는 기준이 될 작은 모델들을 세웁니다. 원본 서버의 작업, 퍼즐 작업, 세션을 통한 비용 분할 상환, 그리고 신뢰할 수 없는 클라이언트가 서버에 붙잡아 둘 수 있는 메모리입니다.

학습 목표 이 장을 마치면 게이트가 제 비용보다 큰 이득을 내는지 판정하는 부등식을 쓰고, 작업 증명이 입증하는 것과 입증하지 않는 것을 말하며, 해시 탐색의 기댓값과 꼬리를 유도하고, 세션이 작업 비용을 분할 상환하는 방식을 설명하고, 이 책의 나머지 부분이 지키는 다섯 가지 불변식을 나열할 수 있어야 합니다.

요청 허용은 경제적 결정입니다

원본 서버가 허용된 요청마다 데이터베이스를 조회하고 페이지를 렌더링한다고 합시다. 그 작업을 c_o, 요청당 게이트의 작업을 c_g, 도착률을 r이라고 둡니다. 게이트가 없으면 원본 서버는 초당 rc_o 단위의 작업을 감당해야 합니다. 비율 a만큼 허용하는 게이트는 그 수요를 rac_o로 바꾸고, 자신은 rc_g를 씁니다. 게이트는 절약한 원본 작업이 게이트 자신의 비용과 정상 방문자에게 지우는 비용을 넘을 때에만 쓸모가 있습니다.

$$rc_g + rac_o < rc_o \quad \Leftrightarrow \quad c_g < (1-a)c_o .$$

이것은 용량 모델이며 측정이 아닙니다. 대역폭, 연결 상태, 저장소, 클라이언트 지연은 빠져 있습니다. 이 모델의 가치는 빠진 항이 모두 눈에 보인다는 데 있습니다. 캐시 적중은 c_o를 작게 만들 수 있고, 비싼 쿼리는 크게 만들 수 있습니다. 요청자 대 서버의 보편적인 비용 비율은 없습니다.

풀이 예제: 게이트가 이득이 되는 경우 $c_o = 100$ 작업 단위, $c_g = 1$로 둡니다. 요청의 절반이 거부되면, 게이트를 둔 시스템은 도착 하나당 100 대신 $1 + 0.5 \times 100 = 51$ 단위를 씁니다. 천 개 중 하나만 거부되면 100.9를 씁니다. 이 모델에서는 게이트가 절약하는 것보다 더 많은 비용이 듭니다. 이 숫자들은 임의로 고른 값이며 측정한 시간이 아닙니다. 모든 성능 주장에 워크로드를 함께 밝혀야 하는 이유를 보여 줄 뿐입니다.

작업 증명이 입증하는 것

클라이언트 퍼즐은 누군가 공개된 검증 규칙을 만족하는 입력을 찾았다는 사실을 입증합니다. 상대가 사람이라는 사실, 신원, 선한 의도는 입증하지 않습니다. 요청자는 연산 자원을 빌리거나, 유효한 세션을 수명 안에서 재사용하거나, 작업을 여러 기계에 나눌 수 있습니다. 퍼즐은 요

청 허용의 비용을 바꿉니다. 허용된 요청이 무엇을 할 수 있는지는 여전히 정책과 검사가 결정합니다.

주소 기반 제한은 이를 보완합니다. 주소 기반 제한은 주소 하나를 제약하지만, 공유 주소 하나가 많은 사람을 나타낼 수 있고 요청자 하나가 많은 주소를 쓸 수 있습니다. 퍼즐도 주소도 신원을 판정해 주지 못합니다. 그래서 Sibuna는 요청 허용, 검사, 평판을 서로 다른 결정으로 유지합니다.

확률변수로 보는 해시 탐색

이상적인 256비트 다이제스트에서 선행 0비트 b개를 요구하면, 독립 시행마다 성공 확률은 $p = 2^{-b}$입니다. 첫 성공까지의 시행 횟수를 K라고 하면 다음이 성립합니다.

$$P(K = k) = (1-p)^{k-1}p, \quad P(K > k) = (1-p)^{k}, \quad E[K] = 1/p = 2^{b}.$$

기댓값은 꼬리 합에서 나옵니다.

$$E[K] = \sum_{k=0}^{\infty} P(K > k) = \sum_{k=0}^{\infty} (1-p)^{k} = 1/p.$$

난이도를 한 단계 올리면 기대 시행 횟수가 두 배가 됩니다. 그렇다고 모든 퍼즐이 두 배 오래 걸린다는 약속은 아닙니다. 어떤 퍼즐은 첫 시행에서 성공하고, 어떤 퍼즐은 평균보다 훨씬 오래 걸립니다. 풀이 한 번만 측정한 클라이언트는 대부분 이 무작위성을 측정한 셈입니다.

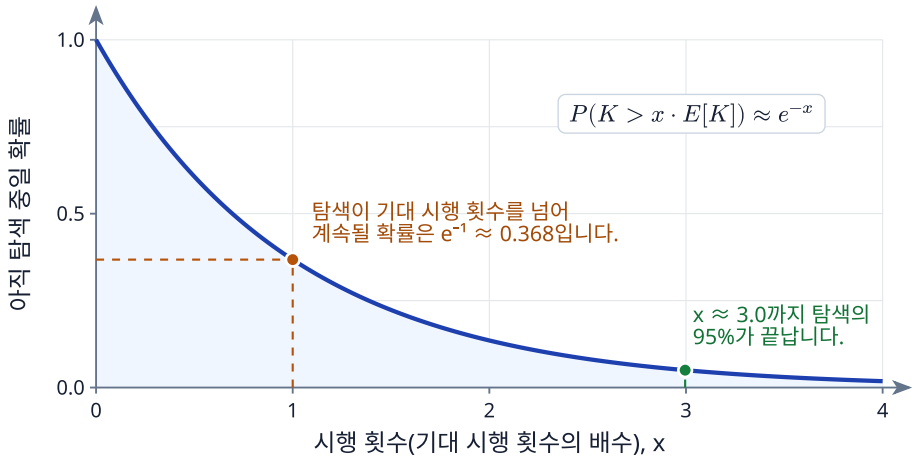


그림 1 해시 탐색의 생존 확률: 기대 시행 횟수의 x배를 시행한 뒤에도 탐색이 아직 진행 중일 확률입니다. 작업량이 클 때의 근사 $P(K > x/p) \approx e^{-x}$를 사용합니다.

풀이 예제: 기댓값은 마감 시간이 아닙니다 $b = 16$에서 기대 작업량은 해시 65,536회입니다. 그만큼 시행한 뒤에도 아직 탐색 중일 확률은 약 $e^{-1} = 0.368$입니다. 95번째 백분위수 시행 횟수는 $\lceil \ln(0.05)/\ln(1-2^{-16}) \rceil \approx 196327$로, 평균의 거의 세 배입니다. 사용자 인터페이스는 기대 완료 시간에 실패를 알리지 말고 이 정도의 편차를 감안해야 합니다.

세션은 작업 비용을 분할 상환합니다

세션 하나가 만료 전까지 요청 m개를 허용하고, 퍼즐 비용이 c_p, 검증 비용이 c_v, 세션 확인 비용이 c_s라고 합시다. 실패한 시도를 무시하면 분할 상환한 요청당 게이트 비용은 $c_v/m + c_s$이고, 클라이언트의 퍼즐 비용은 c_p/m입니다. 세션 수명을 늘리면 사람과 자동화 클라이언트 모두에게 똑같이 도움이 됩니다. 세션 수명을 정하는 일은 정책 결정이며 암호학적 최적화가 아닙니다.

연습 문제 1.1. 클라이언트가 세션 하나로 요청 100개를 보내고, 시행 백만 회짜리 퍼즐을 풉니다. 분할 상환한 요청당 작업량은 얼마일까요? 이 클라이언트가 두 번째 프로세스와 세션을 공유하면 무엇이 달라질까요?
힌트: 비용 계산 모델과 토큰에 실제로 묶인 값을 구별하세요.

제한된 상태는 두 번째 예산입니다

작업을 요청 경로 밖으로 옮긴다고 작업이 사라지지는 않습니다. 인시던트가 비율 λ로 도착하고, 저장소 스레드가 평균 비율 μ로 이를 영속화하며, 큐가 레코드 Q개를 담는다고 합시다. $\lambda > \mu$일 때 유체 근사로 다음을 얻습니다.

$$t_{\text{fill}} \approx \frac{Q}{\lambda - \mu}.$$

큐가 크면 시간을 벌 수 있지만 처리량을 늘려 주지는 않습니다. 일괄 처리는 트랜잭션 비용을 나누지만, 보관된 레코드는 여전히 메모리를 차지하고 결국 처리되어야 합니다. 과부하에서 Sibuna는 새 인시던트 레코드를 버리고 그 수를 셉니다. 요청에 대한 판단은 메모리 상태만으로 계속 내립니다. 운영자는 HTTP 성공률뿐 아니라 잃어버린 증거도 감시해야 합니다.

연습 문제 1.2. 큐 하나가 레코드 512개를 담습니다. 도착률은 1,000/s이고 배출률은 600/s입니다. 처음에 비어 있던 큐는 초과분을 얼마 동안 흡수할 수 있을까요? 이 답이 근사일 뿐인 이유는 무엇일까요?
힌트: 두 비율의 차를 쓰고, 그다음 버스트와 일괄 커밋을 생각해 보세요.

앞으로 지켜 갈 불변식

1. 신뢰할 수 없는 요청은 서버 상태를 무한정 늘릴 수 없습니다.
2. 유효한 요청 허용 토큰도 애플리케이션의 거부를 우회하지 못합니다.
3. 버퍼는 그 버퍼에서 빌린 모든 슬라이스가 살아 있는 동안 살아 있습니다.
4. 영속 재시도도 인시던트나 그 평판 효과를 여러 번 반영할 수 없습니다.
5. 게시된 정책 스냅숏은 마지막으로 읽는 쪽이 놓아줄 때까지 바뀌지 않습니다.

이 책의 나머지 부분은 이 명제들을 참으로 만드는 메커니즘과, 위반을 드러낼 테스트를 이끌어 냅니다. 빠른 경로는 이 명제들이 참인 동안에만 쓸모가 있습니다.

불변식 설명하기. 게이트 부등식 $c_g < (1-a)c_o$를 말로 풀어 쓰고, 측정한 검증 지연 시간만으로는 운영자가 게이트를 운영할 가치가 있는지 알 수 없는 이유를 설명하세요.

제I I부

선행 연구와 설계 공간

먼저 *Sibuna*가 배워 온 시스템들 사이에서 *Sibuna*의 자리를 찾습니다. 암호학 문헌의 클라이언트 퍼즐과 순차 작업, 시그니처 기반 및 의미 기반 애플리케이션 방화벽, 작업 증명 중간 페이지, 호스팅형 엣지 네트워크가 그 시스템들입니다. 그런 다음 운영자가 서로 견주어 볼 가능성이 가장 큰 네 도구를 기능별로 비교하고, 어떤 사실을 직접 확인했는지 기록합니다.

Sibuna의 뿌리

학습 목표 이 장을 마치면 1992년의 클라이언트 퍼즐부터 2018년의 순차 작업 구성까지 이어지는 계보를 추적하고, Sibuna가 결합한 세 가지 아키텍처 아이디어를 꼽고, Sibuna, Anubis, SafeLine, Cloudflare WAF를 하나의 설계 공간에 놓고, 어떤 비교 주장을 검증했고 어떤 주장은 문서에서 읽기만 했는지 말할 수 있어야 합니다.

클라이언트 퍼즐과 순차 작업

Dwork와 Naor의 "Pricing via Processing"(CRYPTO 1992)은 서비스가 요청자를 위해 일하기 전에, 적당히 어렵고 확인하기 쉬운 계산을 요구하자고 제안했습니다. Back의 Hashcash(1997년, 2002년에 논문으로 정리)는 퍼즐을 난이도가 정수 하나로 정해지는 부분 해시 역산으로 만들었고, Juels와 Brainard의 클라이언트 퍼즐(NDSS 1999)은 이 아이디어를 연결 폭주에 적용했습니다. 연결 폭주에서는 서버가 풀리지 않은 퍼즐에 대해 아무것도 기억해서는 안 됩니다. 세 연구 모두 Sibuna가 이어받은 형태를 공유합니다. 서버가 고른 명제, 비용을 조절할 수 있는 풀이, 그리고 풀이보다 몇 자릿수 값싼 검증입니다.

해시 탐색은 완전히 병렬화할 수 있으므로, 코어가 많은 요청자가 더 빨리 끝냅니다. Mahmoody, Moran, Vadhan(CRYPTO 2013)은 순차 작업 증명을 정의했습니다. 증명자는 코어 수와 상관없이 서로 의존하는 긴 단계의 사슬을 수행해야 합니다. Cohen과 Pietrzak(EUROCRYPT 2018)은 검증자가 $O(t \log N)$의 작업만 하면 되는 간단한 해시 그래프 구성을 내놓았습니다. Blocki, Lee, Zhou(ITC 2021)는 양자 랜덤 오라클 모델에서 같은 구성이 양자 증명자에 대해서도 건전함을 증명했습니다. 제III부는 이 연구 흐름에서 Sibuna의 두 번째 계층을 유도하며, SID 0006은 그 가정을 기록합니다.

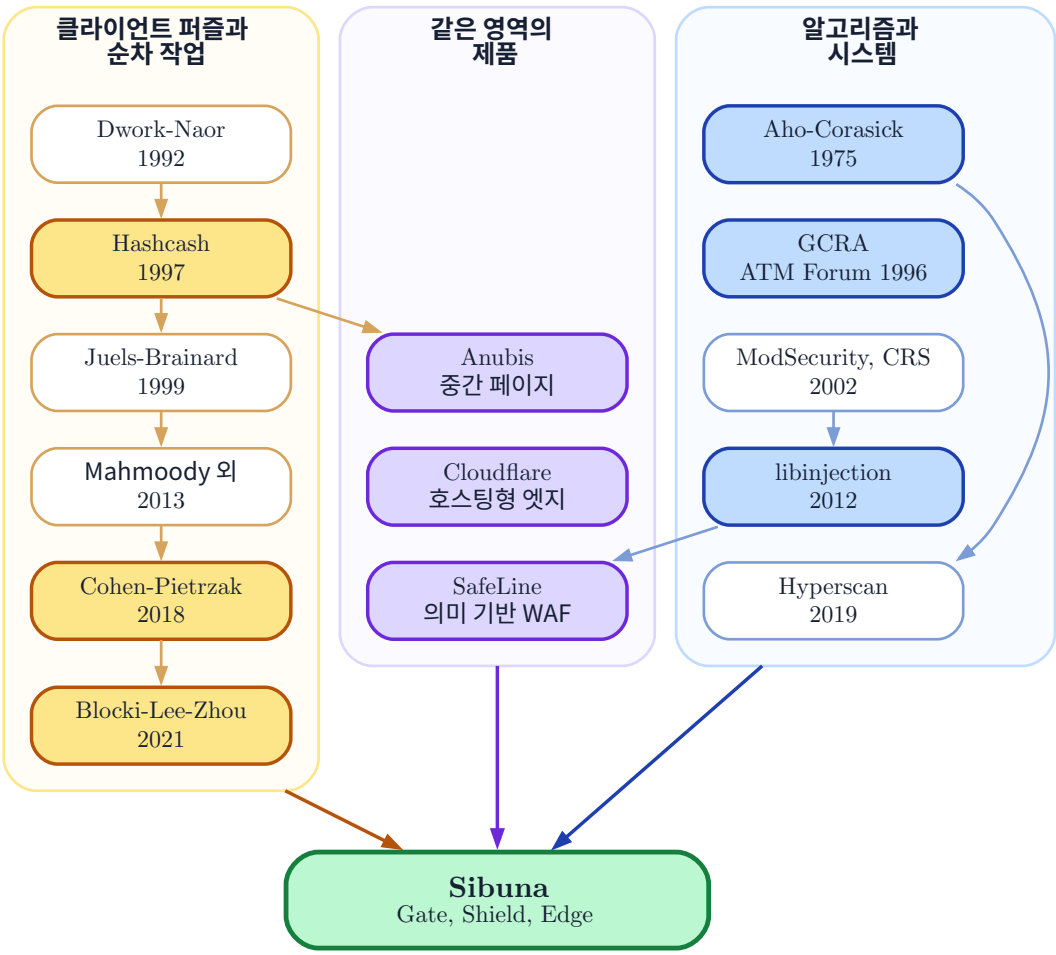


그림 2 Sibuna가 결합한 아이디어의 계보. 왼쪽 열은 클라이언트 퍼즐과 순차 작업의 암호학, 오른쪽 열은 Sibuna가 바탕으로 삼은 알고리즘과 시스템, 가운데 열은 같은 영역의 제품입니다. 시간은 위에서 아래로 흐릅니다. 가는 화살표는 직접적인 계승을 뜻하고, 색이 채워진 상자는 Sibuna가 직접 가져온 아이디어입니다. 이전 저장소 설계는 Multi-Paxos(Lamport 1998; SID 0005)를 바탕으로 했습니다. 지금 Sibuna는 상태를 SQLite나 PostgreSQL에 저장합니다(SID 0011).

애플리케이션 방화벽

1세대 웹 애플리케이션 방화벽은 요청 바이트를 정규 표현식과 대조했습니다. ModSecurity(2002)와 OWASP Core Rule Set은 이 방식을 공개하고 널리 퍼뜨렸으며, 그 비용도 널리 알렸습니다. 요청마다 백트래킹하는 패턴 수백 개, 손으로 조정하는 이상 점수, 평범한 입력에서 끝없이 이어지는 오탐이 그것입니다. libinjection(Hanson, 2012)은 패턴 목록 대신 문자열이 SQL로 파싱되는지 묻는 토크나이저를 써서 비용과 오탐을 모두 줄였습니다. Chaitin의 SafeLine은 이 의미 기반 아이디어를 완전한 제품으로 옮겼고, Hyperscan(Wang 외, NSDI 2019)은 입력이 길 때 벡터화된 리터럴 매칭을 어디까지 밀어붙일 수 있는지 보여 주었습니다. 제VI부는 Sibuna가 검사하는 짧은 필드에 정규 표현식이나 SIMD 엔진 대신 밀집 오토마톤과 작은 구조 토크나이저를 고른 이유를 설명합니다.

중간 페이지와 엣지

웹사이트 앞에 두는 작업 증명 중간 페이지는 2025년에 흔해졌습니다. 언어 모델 학습 데이터를 모으는 스크레이퍼 트래픽 때문에 평범한 원본 서버도 운영 비용을 감당하기 어려워졌기 때문입니다. Anubis(Techaro)는 가장 잘 알려진 공개 구현입니다. JavaScript 퍼즐을 제공하고, 세션 토큰에 서명하며, 그 밖의 트래픽은 그대로 통과시키는 Go 리버스 프록시입니다. Sibuna의 Gate 구성은 상태 없는 챌린지, 대칭 토큰, 두 번째 작업 계층으로 같은 문제를 풉니다. 아래 비교는 둘이 어디서 다른지 기록합니다.

규모의 반대쪽 끝에서는 Cloudflare 같은 호스팅형 엣지가 전역 제어 평면을 두고 수백 개의 거점에 검사, 속도 제한, 봇 점수를 배치합니다. Sibuna의 Edge 배포는 그 형태(노드마다 불변 스냅숏을 공급하는 공유 정책·평판 평면)를 빌리면서도 셀프 호스팅을 유지합니다. 그래서 저장소 계층은 서비스 호출이 아니라 운영자가 직접 운영하는 데이터베이스(노드 하나에서는 SQLite 파일, 클러스터에서는 공유하는 PostgreSQL 데이터베이스 하나)이며, 요청 경로 밖의 저장소 스레드가 이를 읽습니다.

하나의 바이너리에 담은 세 가지 아이디어

Sibuna는 요청 허용을 위한 브라우저 작업 챌린지, 범위가 제한된 애플리케이션 페이로드 검사, 그리고 로컬 정책 스냅숏을 게시하는 공유 제어 평면을 결합합니다. Gate와 Shield는 두 가지 운영 구성이며, 저장소와 분산 배포는 어느 쪽에서든 선택할 수 있습니다.

속성	Gate	Shield
요청 허용	네이티브 Hashcash 또는 순차 작업 검증	같음
세션	키 기반 BLAKE3, 선택적으로 Ed25519	같음
로컬 제어	규칙, CIDR, GCRA, 차단	같음
검사	꺼짐	범위가 제한된 시그니처와 구조 토크나이저
상태	선택적인 영속 정책과 평판(클러스터 전체에서 공유)	검사 포렌식도 포함

브라우저 솔버는 네이티브 검증자와 같은 증명 소스에서 컴파일됩니다. 밀집 리터럴 매칭은 백트래킹을 피하고 스캔 작업을 입력 길이에 선형으로 유지합니다. 불변 정책 스냅숏은 요청 처리를 SQL과 데이터베이스의 가용성에서 분리합니다. 그렇다고 이 가운데 어느 선택도 보편적인 알고리즘 최적성을 보장하지는 않습니다. 짧은 헤더, 긴 본문, 패턴의 다양성, 코어 수, 캐시 크기에 따라 절충점이 달라집니다.

> **채택한 설계 제약**
>
> 1. 요청 경로에서 힙 할당을 하지 않습니다. 모든 테이블에는 고정된 용량과 명시된 포화 시 동작이 있습니다.

2. 챌린지를 발급해도 아무것도 기록하지 않습니다. 대가를 치른 풀이만 메모리를 차지합니다.
3. 세션은 요청을 허용할 뿐, 검사나 명시적 거부에서 요청을 면제하지는 않습니다.
4. 주장하는 모든 숫자는 커밋된 하니스로 측정했거나 모델이라고 표시합니다.
5. 암호학적 선택은 발표 학회와 증명을 인용하고, 양자 공격에 대한 안전성 입장을 밝힙니다.

기능 비교

아래 네 표는 Sibuna와 함께 자주 언급되는 세 도구를 Sibuna와 비교합니다. 다른 제품에 관한 사실은 2026년 9월에 공개 문서와 릴리스 파일에서 읽었습니다. Anubis 1.27.0(MIT 라이선스, Go), SafeLine 커뮤니티 에디션 9.x(비공개 탐지기 이미지를 둘러싼 GPL-3.0 관리 코드, Docker Compose), Cloudflare WAF 개발자 문서입니다. Anubis는 벤치마크 호스트에서도 실행했고, SafeLine과 Cloudflare는 실행하지 않았습니다(이유는 제VIII부에서 설명합니다). 대시는 제품이 그 기능을 제공하지 않음을, 요금제 이름은 기능이 있지만 그 요금제에서만 제공됨을 뜻합니다. Sibuna의 각 측정값은 그 값이 나온 커밋된 결과 파일을 밝힙니다. TLS 지문 행은 2026년 10월에 Anubis 변경 기록(1.21.0, 1.26.0), SafeLine 릴리스 노트(8.10.0, 9.2.0), Cloudflare의 "JA3/JA4 fingerprint" 페이지와 대조해 확인했습니다.

표 3 기능 비교 1부: 배포

기능	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
실행 형태	바이너리 하나; 리버스 프록시 또는 forward-auth; 선택적으로 SQLite 파일 또는 공유 PostgreSQL 데이터베이스	Go 바이너리 하나; 리버스 프록시 또는 forward-auth	Docker 컨테이너 일곱 개 (tengine, detector, mgt, luigi, fvm, chaos, PostgreSQL)	호스팅형 네트워크; DNS가 Cloudflare를 가리킴
언어, 런타임	Rust, 가비지 컬렉터 없음; 저장소를 끄면 외부 crate 없음	Go, 가비지 컬렉션 사용	C/C++ 프록시와 탐지기, Go 관리 코드, PostgreSQL	독점 엣지 소프트웨어
라이선스	LGPL-3.0-or-later	MIT	GPL-3.0 관리 코드; 비공개 탐지기 이미지	독점 서비스
호스트 자원 사용량	바이너리 35.7 MiB(`benchmarks/results/comparison-latest.json`); 유휴	바이너리 38.1 MiB; 유휴 RSS 21 MiB; Linux	최소 CPU 코어 1개, RAM 1 GB, 디스크 5	온프레미스에는 없음

기능	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
	RSS 5.9 MiB(`benchmarks/results/primitives-latest.json`); 부하 시 최대 RSS 16.0 MiB 이하(`benchmarks/results/product-latest.md`)	테스트 환경, 콘솔 꺼짐	GB, Docker 20.10 이상	

표 4 기능 비교 2부: 요청 허용과 세션

기능	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
작업 증명 요청 허용	Hashcash(비트 단위)와 Cohen-Pietrzak 순차 작업, 규칙마다 선택; WebAssembly와 JavaScript 대체 경로	16진수 니블 단위의 SHA-256 Hashcash(`fast`, `slow`); 작업이 없는 `metarefresh`와 `preact` 챌린지	JavaScript 봇 방지 챌린지와 CAPTCHA; 작업 증명 방식 아님	관리형 챌린지, JS 챌린지, Turnstile; Bot Fight Mode는 계산 챌린지를 발급
서버의 챌린지 상태	풀리기 전에는 없음; 풀린 태그는 고정 Robin Hood 테이블에 저장	저장소 백엔드: memory, bbolt, Valkey, S3 중 하나	스택이 관리	Cloudflare가 관리
세션 토큰	키 기반 BLAKE3 태그, 치른 작업에 묶인 75자 쿠키; 선택적으로 Ed25519	Ed25519로 서명한 JSON Web Token; 선택적으로 HS512	스택이 발급하는 쿠키	`cf_clearance` 쿠키
기본 토큰의 양자 내성	대칭 방식: Grover 알고리즘의 이차 가속만 적용됨	Ed25519: Shor 알고리즘으로 깨짐	문서화되지 않음	문서화되지 않음
정책 언어	JSON 규칙: 경로, User-Agent, 헤더, IPv4/IPv6 CIDR, JA4 지문; 임계값이 있는 ALLOW, DENY, CHALLENGE,	YAML/JSON 봇 정의: UA, 경로, 헤더에 대한 정규식; CIDR; CEL 표현식; 임계값이 있는 WEIGH	콘솔에서 정의하는 사용자 지정 규칙, ACL, IP 그룹	Rules language 표현식; 요금제별 사용자 지정 규칙 *5 / 20 / 100 / 1000*개

기능	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
	WEIGH; 데이터베이스의 동적 규칙			
TLS 지문 (JA4)	규칙이 JA4 지문을 정확히 일치 또는 접두사로 비교; 신뢰할 수 있는 인그레스가 지문을 계산해 헤더로 전달(`--ja4-header`); Sibuna는 직접 계산하지 않음	문서화되지 않음; 정책에서 HTTP 지문 JA4H를 참조할 수 있으며, Anubis가 이를 계산해 `X-Http-Fingerprint-JA4H` 헤더로 추가함	자체 TLS 종료 지점에서 계산; 공격 상세 정보에 표시; 사용자 지정 규칙으로 비교(같음, 같지 않음); 내장 규칙이 자체 악성 JA4 위협 정보에 있는 지문을 거부	WAF 사용자 지정 규칙과 분석에서 JA3, JA4 지문 사용 *(Enterprise, Bot Management* 포함*)*

표 5 기능 비교 3부: 검사와 상태

기능	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
SQLi, XSS, RCE, 경로 탐색 검사	Shield: 태그가 붙은 오토마톤과 구조 토크나이저, 본문의 처음 8 KB	-	OWASP 범주에 대한 의미 분석 엔진, SSRF, XXE, CRLF도 포함	관리형 규칙 집합 *(Pro* 이상*)*; OWASP Core Ruleset; 공격 점수 *(Business* 이상*)*
속도 제한	클라이언트별 GCRA, 노드 로컬	- (CEL 규칙에서 시스템 부하를 참조할 수 있음)	IP, 경로, 세션별	요금제별 규칙 *1 / 2 / 5 / 100* 개; Enterprise 미만은 IP 전용 키
차단과 평판	허니팟 차단; 클러스터 전체가 공유하는 평판 테이블에서 불러온 평판 트라이	DNSBL; Thoth를 통한 ASN과 GeoIP *(유료, 비공개)*	IP 그룹과 차단 목록; 위협 인텔리전스 *(Pro)*	IP 목록; 관리형 목록과 봇 점수 *(Enterprise)*
포렌식	SQLite 또는 PostgreSQL: 인시던트 전문 검색, 벡터 기반 캠페인 군집화	Prometheus 지표만	PostgreSQL 공격 로그와 콘솔	Security Events; Free 요금제에서는 표본만 표시

기능	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
다중 노드	정책과 평판을 위해 모든 노드가 공유하는 PostgreSQL 데이터베이스 하나; 공유 시드; 발급 노드 고정 챌린지 검증	공유 서명 키; 공유 Valkey 저장소	호스트당 스택 하나	전역 애니캐스트

표 6 기능 비교 4부: 운영

기능	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
지표	`/__sibuna/metrics`에서 Prometheus 형식	별도 포트의 Prometheus	콘솔 대시보드	대시보드와 분석
TLS 종료	- (인그레스)	- (인그레스)	**예**	**예**
관리 콘솔	선택 사항; SID 0007 Committed	-	**예**	**예**
벤치마크 호스트에서 측정	**예** (제VIII부)	**예**	- (Docker 전용)	- (호스팅형)

제품을 고를 때는 세 가지 차이가 가장 큰 비중을 차지합니다.

- 작업을 누가 하는가. Anubis와 Sibuna Gate는 원본 서버가 무슨 일이든 하기 전에 클라이언트가 먼저 비용을 치르게 합니다. SafeLine과 Cloudflare는 먼저 검사하고 필요할 때만 챌린지를 제시합니다. 기본 방침은 일단 허용하고 걸러 내는 것입니다. Sibuna Shield는 이 순서로 두 가지를 모두 합니다. 이미 유효한 세션을 가진 요청이라도 검사에서 거부될 수 있습니다.
- 서버가 무엇을 기억하는가. Anubis는 챌린지 상태를 저장소(memory, bbolt, Valkey, S3)에 두고 JSON Web Token에 Ed25519로 서명합니다. Sibuna는 스스로를 인증하는 레코드를 발급하고, 풀린 태그만 고정 테이블에 기억합니다. SafeLine은 공격 로그를 PostgreSQL에 보관합니다. Cloudflare는 모든 것을 보관하지만, Free 요금제에서는 표본만 볼 수 있습니다.
- 운영자가 무엇을 운영해야 하는가. Sibuna는 바이너리 하나에 선택적으로 SQLite 파일, 또는 클러스터가 공유하는 PostgreSQL 데이터베이스가 붙습니다. Anubis는 Go 바이너리 하나입니다. SafeLine은 코어 하나 이상, 메모리 1기가바이트 이상, 디스크 5기가바이트 이상인 Linux 호스트에서 Docker 데몬 위에 실행되는 컨테이너 일곱 개입니다. Cloudflare는 DNS 변경과 구독만 하면 됩니다.

현재 제품의 경계 Sibuna는 인그레스의 TLS 종료를 배포 환경의 프록시에 맡기며, 학습된 모델로 봇 점수를 매기지 않고 유료 시그니처도 제공하지 않습니다. 신뢰할 수 있는 인그레스가 계산한 JA4 지문을 비교할 수는 있지만, 지문을 직접 계산하지는 못합니다(SID 0014, 제IX부). 선택형 콘솔은 인증된 대시보드, 국가 정보 보강, 정책 편집, 조사, 클러스터 관리를 제공합니다. 국가 조회와 저장소 작업은 요청 분류 밖에서 실행됩니다. SID 0007은 Committed 상태이며, 콘솔은 브라우저 워크플로 테스트를 통과합니다(SID 0010). 대시보드 여덟 개를 활성화하고 부하를 건 상태의 공식 성능 격리 관문은 아직 측정하지 않았습니다. 측정한 내용은 제VIII부에 기록되어 있습니다.

연습 문제 2.1. 검증자 소스 하나를 네이티브 코드와 WebAssembly로 함께 컴파일하면 프로토콜의 어긋남이 줄어드는 이유를 설명하세요. 그래도 독립적인 테스트나 암호학적 검토가 필요한 속성은 무엇일까요?

연습 문제 2.2. 비교 표만 사용해서, 정적 사이트에 (a) 작업 증명 게이트, (b) SQL 인젝션 검사, (c) 클러스터 전체 차단을 제공하는 가장 작은 배포를 적어 보세요. 각 선택은 어느 행 때문에 정해졌을까요?
힌트: "실행 형태"와 "다중 노드" 행에서 시작하세요.

불변식 설명하기. Gate, Shield, 선택적 공유 저장소가 요청 허용, 검사, 정책 배포를 어떻게 나누는지 설명하세요. 그런 다음 고객에게 인용하기 전에 다시 검증하고 싶은 비교 표의 행 하나를 고르고, 그것을 어떻게 검증할지 말하세요.

제III부

암호학 연구의 기초

*Sibuna*의 두 작업 증명 계층에 깔린 비용 모델을 유도하고, *Cohen-Pietrzak* 순차 작업 증명과 그 보안 논증을 제시하며, 챌린지와 세션에 키 기반 해시 인증을 쓰는 근거를 밝힙니다. 가정과 참고 문헌은 *SID 0006*에 기록되어 있습니다.

Hashcash: 첫 번째 계층의 수학

학습 목표 이 장을 마치면 비트 단위 Hashcash 퍼즐의 기대 작업량, 분산, 꼬리 확률을 말하고, 서버의 검증 비용이 공격자의 이점을 제한하는 이유를 설명하며, 두 번째 계층의 동기가 되는 두 가지 한계를 설명할 수 있어야 합니다.

정의와 비용

챌린지 문자열 C와 난이도 b비트에 대한 Sibuna의 첫 번째 계층 퍼즐은 다음과 같습니다. SHA-256 $(C \,\|\, ":" \,\|\, \mathrm{dec}\,(N))$의 선행 0비트가 b개가 되는 $N \in \mathbb{N}$을 찾습니다. nonce를 십진수로 표기한 것은 의도적인 선택입니다. 그러면 이진 인코딩 단계 없이도 WebAssembly 솔버, JavaScript 대체 솔버, 네이티브 검증자 사이에서 전송 형식이 똑같이 유지됩니다.

난이도의 단위는 압축 함수 호출이 아니라 기대 해시 시행입니다. 입력 하나가 SHA-256 블록 여러 개를 차지할 수 있습니다. 접두사를 미리 계산하면 nonce당 작업량이 달라질 수도 있습니다. 전송 규칙은 여전히 다이제스트에 대한 술어입니다. 구현 사이에서 일치해야 하는 것은 추정한 CPU 비용이 아니라 명제와 nonce 인코딩입니다.

독립적인 이상적 다이제스트에서 시행 횟수 X는 $p = 2^{-b}$인 기하 분포를 따릅니다. 중앙값은 $\lceil \ln(1/2) / \ln(1-p) \rceil$로, 작은 p에서는 약 $2^b \ln 2$이고, 분산은 $(1-p)/p^2$입니다. 검증자는 탐색을 수행하지 않고 후보 다이제스트 하나를 확인합니다. 파싱, 챌린지 인증, 지문 확인, 사용 완료 상태 삽입, 응답은 추가 작업입니다. 기본 연산 하나의 측정 시간을 역산해 서버가 견딜 수 있는 폭주 트래픽 용량을 구할 수는 없습니다.

탐색의 두 가지 한계

첫째는 분산입니다. 탐색이 오래 걸린다고 워커가 고장 난 것은 아닙니다. 둘째는 병렬성입니다. 독립적인 nonce 시행은 동시에 실행할 수 있습니다. 기대 총 시행 횟수가 같아도 코어가 많으면 탐색의 경과 시간이 줄어듭니다. 그래서 요청 허용 정책에서는 하드웨어 처리량이 중요합니다. 누군가 내세우는 해시율이나 벤치마크 하나로는 모든 요청자의 이점에 상한을 둘 수 없습니다.

연습 문제 3.0. 기하 분포의 꼬리를 써서 시행이 4.6×2^b회보다 많이 필요할 확률을 추정하세요. nonce 공간을 여러 워커에 나누어도 기대 시행 횟수가 변하지 않는 이유는 무엇일까요?

힌트: 경과한 초가 아니라 총 시행 횟수를 세세요.

순차 작업 증명: 두 번째 계층

학습 목표 Cohen-Pietrzak 라벨링, 이 라벨링이 순차 계산을 강제하는 이유, 검증자가 확인하는 내용, 조건부 표본 추출 상한 $(1-\alpha)^t$와 그 한계, 포스트 양자 환경에서의 결과, 증명자의 메모리 상한, 그리고 작업 비트 대응의 근거가 된 보정 데이터를 이해합니다.

순차 작업을 쓰는 이유

순차 작업 구성은 계산 사이에 의존성을 둡니다. 나중 라벨을 계산하려면 앞서 만든 값이 필요합니다. 이 구성이 다루는 양은 총 작업량이 아니라 계산의 깊이입니다. 독립적인 퍼즐은 여전히 병렬로 실행할 수 있고, 하드웨어의 해시 지연 시간도 여전히 중요합니다.

Sibuna는 Cohen과 Pietrzak가 설명한 해시 기반 구성을 따릅니다. 깊이 n에서 그래프에는 라벨이 $2^{n+1}-1$개 있지만, 라벨 수가 고정되었다고 브라우저 지연 시간이 결정되는 것은 아닙니다. 라벨은 압축 블록 여러 개에 걸칠 수 있고, 개방은 부분 트리를 다시 계산할 수 있으며, 스케줄링은 변동을 더합니다. 오라클 모델 분석에서 얻은 보안 명제를 모든 구체적인 구현이나 공격자에 대한 무조건적인 주장으로 바꾸어서는 안 됩니다.

구성

깊이를 n이라 하고, 길이가 n 이하인 이진 문자열로 이름 붙인 노드로 이루어진 완전 이진 트리를 생각해 봅시다(루트 ε, 자식 $v0$와 $v1$, 길이 n의 잎). 각 노드 v에는 32바이트 라벨 $\ell_v = H(\chi, v, \text{parents}\ (v))$가 붙습니다. 여기서 χ는 챌린지 명제이고, H는 χ를 한 번 흡수한 SHA-256입니다(Hashcash 솔버와 같은 접두사 기법).

- 내부 노드는 두 자식을 해시합니다: $\ell_v = H(\chi, v, \ell_{v0}, \ell_{v1})$.
- 잎 $u = u_1 \ldots u_n$은 루트 경로가 오른쪽으로 꺾인 곳마다 조상의 왼쪽 형제를 해시합니다: $\ell_u = H(\chi, u, \{\ell_{u_1 \ldots u_{i-1} 0} : u_i = 1\})$.

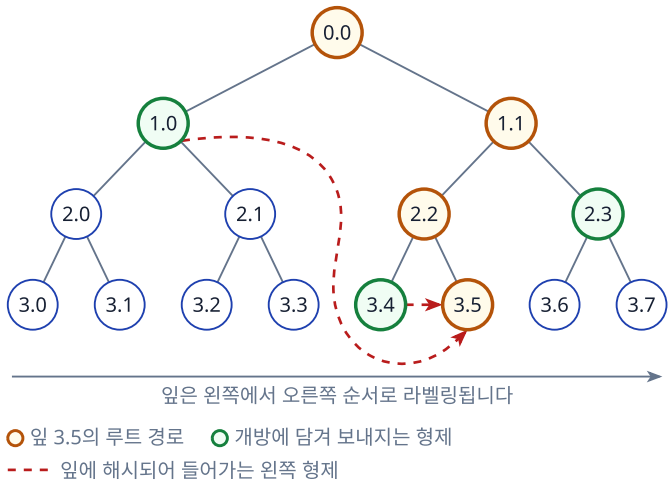


그림 3 깊이 3의 트리: 잎 3.5의 개방은 잎 라벨과 형제 라벨 세 개를 보냅니다. 점선 간선은 라벨링을 순차적으로 만드는 왼쪽 형제 의존성입니다.

계산을 순차적으로 만드는 것은 왼쪽 형제 간선입니다. 잎 u는 그 왼쪽의 모든 부분 트리가 완성되기 전까지 라벨을 붙일 수 없으므로, 깊이 우선 후위 순회가 강제되고 루트 라벨 $\varphi = \ell_\varepsilon$은 $2^{n+1} - 1$개의 라벨 전체에 순서대로 의존합니다.

개방. φ를 계산한 뒤 증명자는 Fiat-Shamir로 잎 t개 $\gamma_i = H(\chi \,\|\, \varphi \,\|\, i) \bmod 2^n$을 도출하고, 각 잎에 대해 잎 라벨과 루트 경로를 따라 놓인 형제 라벨 n개를 보냅니다. 검증자는 잎의 부모인 형제들로 잎을 다시 계산하고, 경로를 따라 위로 해시하며, φ에 도달하는지 확인합니다. 검증자의 비용은 제한된 작업 공간에서 라벨 계산 $O(tn)$번입니다. 라벨 계산 하나가 해시 블록 여러 개에 걸칠 수 있으므로, 라벨 수를 세는 것은 압축 호출 수를 세는 것과 다릅니다.

> **구현: `posw::verify_opening`** 개방된 잎 하나를 형제 라벨로 다시 계산하고 경로를 따라 루트까지 올라갑니다. `crates/pow/src/posw.rs`(crate `sibuna-pow`)에 있습니다.

```
fn verify_opening(base: &Sha256, n: u8, phi: &Label, gamma: u32, opening: &[Label]) -> bool {
    let Some((leaf, siblings)) = opening.split_first() else {
        return false;
    };
    // `siblings`는 깊이 n에서 깊이 1 순서이므로, 뒤집으면 깊이 1..=n과 짝이 맞습니다.
    let mut hasher = node_hasher(base, n, gamma);
    for (d, sibling) in (1..=n).zip(siblings.iter().rev()) {
        if turns_right(gamma, n, d) {
            hasher.update(sibling);
        }
    }
    if hasher.finalize() != *leaf {
        return false;
    }
    let mut current = *leaf;
    for (d, sibling) in (1..=n).rev().zip(siblings) {
        let mut parent = node_hasher(base, d - 1, gamma >> (n - d + 1));
```

```
        if turns_right(gamma, n, d) {
            parent.update(sibling);
            parent.update(&current);
        } else {
            parent.update(&current);
            parent.update(sibling);
        }
        current = parent.finalize();
    }
    current == *phi
}
```

표본 추출, 건전성, 그 한계

독립적으로 탐지할 수 있는 나쁜 비율 α를 가진 고정된 커밋먼트를 생각합니다. 각 개방이 균등하고 독립적으로 표본을 뽑는다면, 개방 t번에서 나쁜 위치를 모두 놓칠 확률은 $(1-\alpha)^t$입니다. 이 기초 계산은 개방이 많을수록 도움이 되는 이유를 설명합니다. 그러나 이것만으로는 순차 작업 프로토콜의 보안 증명이 되지 않습니다.

풀이 예제: 조건부 표본 추출 상한 나쁜 비율이 $\alpha = 0.1$로 고정되고 독립 개방이 $t = 16$번이면, 나쁜 위치를 모두 놓칠 확률은 $0.9^{16} \approx 0.185$입니다. 개방이 32번이면 약 0.034입니다. 이 전제는 표본을 고르기 전에 커밋먼트를 고정합니다. 증명자가 루트를 많이 만들어 골라 볼(grinding) 수 있거나, 어떤 커밋먼트를 공개할지 고를 수 있거나, 실패들을 서로 연관시킬 수 있다면, 이 계산은 그런 전략을 반영하지 못합니다.

참고한 순차 작업 구성에는 오라클 모델에서의 보안 분석이 있습니다. Sibuna의 도메인 분리, 매개변수 대응, 절단, 구현은 여전히 그 구성의 가정에 비추어 확인해야 합니다. Fiat-Shamir 표본 추출은 개방을 커밋먼트로부터 재현할 수 있게 합니다. 재현 가능성만으로는 그라인딩(grinding) 공격에 대한 저항이 증명되지 않습니다. 우리는 속임수가 결코 더 싸지 않다고 무조건 주장하지는 않습니다.

연습 문제 3.1. 공격자가 각각 확률 q로 받아들여지는 독립 커밋먼트 g개를 시도한다면, 적어도 하나가 받아들여질 확률을 유도하세요. 그 식에서 빠진 비용은 무엇일까요?
힌트: 먼저 g개가 모두 실패할 확률을 계산하세요.

증명자의 메모리

단순한 증명자는 모든 라벨($2^{n+1}-1$개)을 저장합니다. Sibuna의 증명자는 첫 번째 패스 동안 현재 경로의 왼쪽 형제 라벨 스택(라벨 $n+1$개)을 유지하고, 상위 $m = \min(n, 10)$개 레벨(라벨 $2^{m+1}-1$개, 최대 64 KB)만 보관합니다. 잎을 개방하려면 그 잎의 깊이 m 조상의 부분 트리를 다시 계산하는데, 이는 해시 $2^{n-m+1}-1$번이며 보관한 레벨로 스택을 채워 시작합니다. 증명자의 총비용은 해시 $2^{n+1}-1+t\,(2^{n-m+1}-1)$번이고, 출력 증명을 포함한 작업 공간 전체가 100 KB 미만입니다. 브라우저 탭과 네이티브 테스트는 같은 `Workspace` 타입(`sibuna_pow::posw::Workspace`)을 공유합니다.

매개변수는 시간이 아닙니다

Sibuna는 작업 비트를 $n = b - 3$으로 깊이에 대응시키고, 지원 구간 $[4, 24]$로 제한합니다. 이는 작업 규모에 관한 관례이며, 알고리즘 사이에 실제 경과 시간이 같다는 보장이 아닙니다. 잎 하나는 부모 라벨 여러 개를 해시할 수 있지만, Hashcash는 nonce를 반복해서 시험합니다. 컴파일러, 브라우저, 프로세서, 매개변수 선택이 모두 상대 비용에 영향을 줍니다. 커밋된 기본 연산 벤치마크는 검증을 측정하며, 브라우저 풀이 지연 시간을 입증하지는 않습니다.

증명 크기는 $32(1 + t(n + 1))$바이트입니다. 깊이 13에 개방 16개이면 $32(1 + 16 \times 14) = 7200$바이트입니다. 트리를 만드는 주된 비용이 그대로여도, 개방 수를 늘리면 증명과 검증자의 작업이 모두 늘어납니다.

연습 문제 3.2. 깊이 17에 개방 16개일 때의 증명 크기를 계산하세요. 그다음 개방 수를 두 배로 늘리세요. 어느 항이 두 배가 되고 어느 항이 그대로일까요?

구성 선택하기

의미 있는 비교는 알고리즘의 평판이 아니라 자원 예산입니다. 증명자의 메모리, 검증자의 최악의 경우 작업량, 증명 바이트 수, 주장하는 속성에 필요한 가정을 기록하세요. 메모리 하드 탐색은 쓸모가 있을 수 있지만, 비싼 작업을 반복해야 하는 검증자는 공격자에게 서버 자원을 소모시킬 또 다른 지점을 줍니다. 공개 검증 가능한 지연 구성은 공유 비밀 기반 게이트와는 다른 신뢰 모델에 쓰입니다.

Sibuna는 값싼 해시 검증과 해시 기반 순차 작업 선택지를 유지합니다. 후자는 증명 바이트가 더 많고 매개변수 분석도 더 신중해야 합니다. 어느 쪽을 택하든, 비싼 검증에 들어가기 전에 형식이 잘못된 증명을 거부해야 한다는 점은 변하지 않습니다.

불변식 설명하기. 라벨이 커밋먼트에 속함을 증명하는 인증 트리와, 라벨을 계산하는 방식을 제약하는 의존 간선의 차이를 설명하세요.

대칭 인증: 챌린지와 세션

학습 목표 발급자와 검증자가 같은 신뢰 도메인에 있을 때 키 기반 의사 난수 함수가 올바른 기본 요소인 이유를 설명하고, 128비트 태그의 위조 상한을 말하며, 상태 없는 챌린지 레코드와 사용 완료 집합의 상한을 설명하고, 키 스케줄를 읽습니다.

발급자가 곧 검증자

디지털 서명은 공개 키를 가진 누구나 검증할 수 있게 하므로, 검증자가 토큰을 발급할 수 없어야 할 때 알맞은 도구입니다. Sibuna의 검증자는 발급자 자신입니다(또는 시드 하나를 공유하는 클러스터입니다). 의사 난수 함수로 만든 메시지 인증 코드는 그 신뢰 도메인 안에서 인증을 제공합니다. 공개 검증은 제공하지 않습니다. MAC 키를 가진 사람은 누구나 토큰을 발급할 수도 있습니다. 벤치마크 장은 구현된 토큰 연산을 비교합니다.

보조정리 3 (위조 상한) 키 모드의 BLAKE3를 PRF로 모델링하고 태그가 16바이트이면, 검증 질의를 q번 하는 공격자가 유효한 토큰을 위조할 확률은 최대 $q \cdot 2^{-128} + \varepsilon_{\mathrm{PRF}}$입니다. 초당 질의가 백만 번이면 초당 2^{-108}입니다.

장기적으로는 두 번째 결과가 중요합니다. Ed25519는 타원 곡선 군의 이산 로그에 기대는데, Shor 알고리즘이 이를 깨뜨립니다. 키 기반 해시는 해시 함수에만 기대며, 양자 알고리즘은 해시 함수에 대해 기껏해야 이차 속도 향상만 줍니다. Sibuna는 토큰을 발급할 수 없는 검증자가 필요한 배포를 위해 Ed25519를 선택지(`--token-scheme ed25519`)로 유지합니다.

키 스케줄

32바이트 마스터 시드 하나(`--secret-file`, `SIBUNA_SECRET`, 또는 새로 만든 무작위 값)에서 키 기반 BLAKE3와 도메인 문자열로 모든 용도별 키를 도출합니다. 토큰 키, 챌린지 키, Ed25519 키, 지문 키, 그리고 요구 티켓을 봉인하는 요구 키입니다(제IV부).

구현: `Keys::derive` 시드에서 용도별 키 다섯 개를 도출합니다. `crates/challenge/src/keys.rs`(crate `sibuna-challenge`)에 있습니다. `Keys`와 `Seed`는 해제될 때 바이트를 0으로 지우며, `Debug` 출력에서는 값을 가립니다.

```
pub fn derive(seed: &Seed) -> Keys {
    Keys {
        token: subkey(seed, "sibuna/token/v1"),
        challenge: subkey(seed, "sibuna/challenge/v1"),
        ed25519_seed: subkey(seed, "sibuna/ed25519/v1"),
        fingerprint: subkey(seed, "sibuna/fingerprint/v1"),
        requirement: subkey(seed, "sibuna/requirement/v1"),
    }
}

fn subkey(seed: &Seed, purpose: &str) -> [u8; 32] {
    blake3::keyed_hash(seed.as_bytes(), purpose.as_bytes())
}
```

클러스터에서는 `Keys::bind_node`가 챌린지 키를 바꿉니다. 새 키는 기존 챌린지 키로 `sibuna/challenge-node/v1`과 노드 id를 키 기반 BLAKE3로 해시한 값입니다. 그래서 챌린지는 발급한 노드로 돌아와야 하지만, 세션 토큰과 요구 티켓은 모든 구성원에서 유효합니다.
도메인 분리는 한 키가 실수로 여러 용도에 재사용되는 일을 막습니다. 마스터 시드가 노출되면 도출된 모든 키가 노출됩니다. 클러스터는 시드 하나에만 합의하면 됩니다.

상태 없는 챌린지

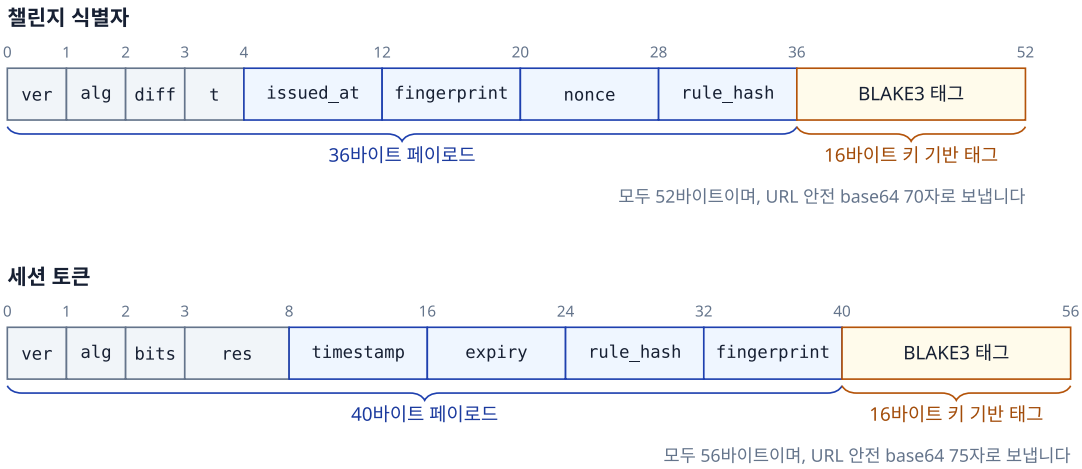


그림 4 챌린지 식별자와 세션 토큰의 전송 형식. 각 레코드 위의 숫자는 바이트 오프셋입니다.

챌린지 식별자는 스스로를 인증하는 레코드입니다. 버전, 알고리즘, 난이도, 개방 수, 발급 시각, 클라이언트의 키 기반 지문, PRF로 도출한 nonce, 챌린지를 요구한 정책 규칙의 해시가 차례로 오고, 그 뒤에 챌린지 키로 만든 16바이트 태그가 붙습니다. 발급할 때는 아무것도 쓰지 않습니다. 검증은 레코드를 디코딩하고, 태그를 상수 시간에 확인하고, 경과 시간을 챌린지 TTL과 비교하고, 지문을 제출한 클라이언트와 비교한 다음에야 증명을 살펴봅니다.

보조정리 4 (상태는 대가를 치른 작업으로만 늘어남) 데몬이 기억하는 챌린지 태그의 집합을 S라고 합시다. 태그는 그에 대한 유효한 증명이 받아들여진 뒤에만 S에 들어가고, 챌린지 TTL이 지나면 빠집니다. 따라서 그 구간에서 서로 다른 풀이가 받아들여지는 속도의 상한이 r이면 $|\,S\,| \le r \cdot \mathrm{TTL}$입니다. 비용 없이 챌린지를 가져오기만 해서는 태그가 할당되지 않습니다. 구현에는 고정 용량도 있어서, 가득 차면 유효한 풀이를 거부할 수 있습니다. 이것은 상태에 대한 상한이며, 모든 공격자의 계산 전략에 대한 하한이 아닙니다.

사용 완료 집합은 16바이트 태그를 담는 Robin Hood 개방 주소법 샤드 16개입니다(제V부). 지문은 클라이언트 주소와 User-Agent의 키 기반 해시입니다. 길이를 붙여 구분하므로 이어 붙여도 서로 다른 두 입력이 충돌하지 않으며, 키를 쓰므로 다른 클라이언트의 지문을 오프라인으로 계산할 수 없습니다.

연습 문제 3.3. 토큰 페이로드에는 작업 수준, `timestamp`, `expiry`, `rule_hash`, `fingerprint`가 담깁니다. 각 필드를 없애면 무엇을 막지 못하게 되는지, 그리고 태그가 다섯 필드를 모두 덮어야 하는 이유를 설명하세요.

불변식 설명하기. 동료가 "더 강하니까" 모든 토큰을 Ed25519로 바꾸자고 제안합니다. 신뢰 도메인 논거, 비용 차이, 포스트 양자 측면의 고려 사항을 설명하세요.

제I V부

Sibuna 프로토콜

구현된 그대로의 전송 프로토콜을 명세합니다. 중간 페이지 왕복, 상태 없는 챌린지 레코드, 두 가지 풀이 형식, 검증 순서, 일회 사용 강제, 세션 쿠키를 다룹니다.

챌린지 왕복

학습 목표 이 장을 마치면 브라우저가 첫 요청에서 세션을 발급받기까지의 과정을 추적하고, 관련된 모든 엔드포인트와 JSON 필드를 말하며, 챌린지가 보호된 경로의 정책 결정을 담고 다니는 이유를 설명할 수 있어야 합니다.

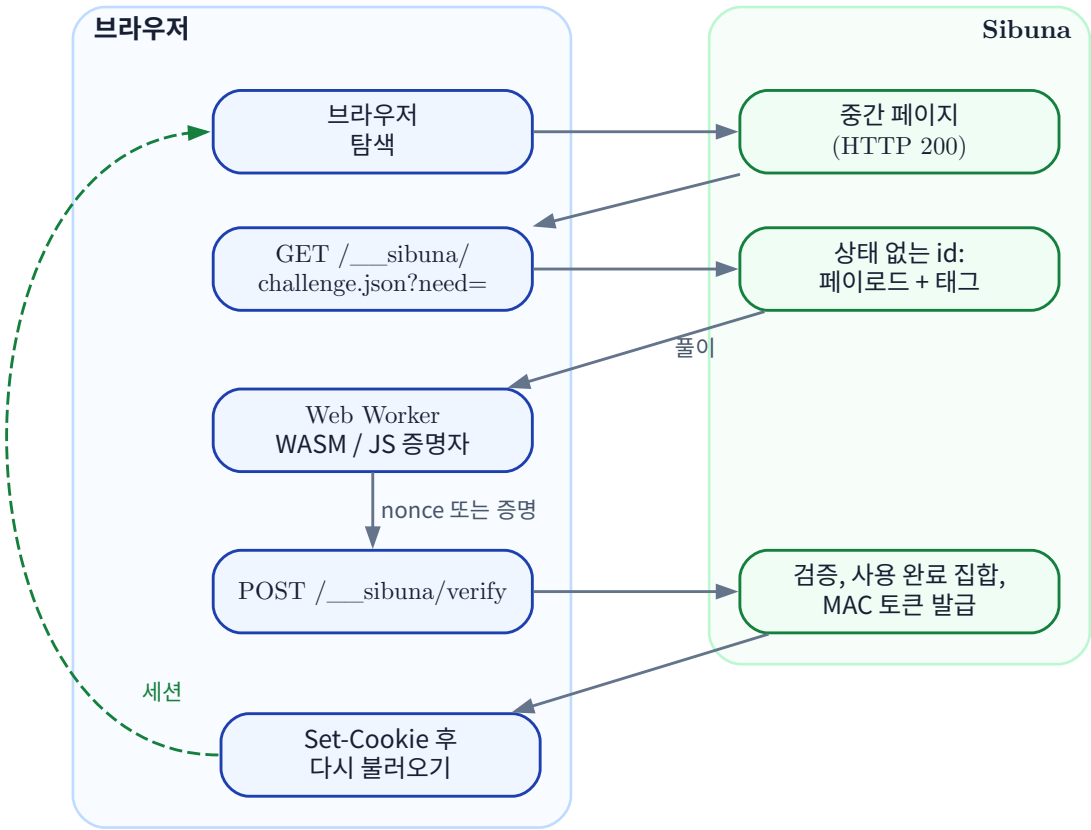


그림 5 챌린지 왕복. 왼쪽 단계는 브라우저에서, 오른쪽 단계는 Sibuna에서 실행됩니다.

1. 유효한 세션이 없는 탐색 요청은 정책 엔진에 도달해 `CHALLENGE`로 분류됩니다. 요청이 `text/html`을 받아들이므로 데몬은 중간 페이지와 `X-Sibuna-Status: CHALLENGE` 헤더를 담아 `200`으로 응답합니다. HTML을 받아들이지 않는 API 클라이언트는 바로 가져올 수 있는 `challenge` URL이 담긴 JSON 본문과 함께 `401`을 받습니다. 어느 쪽이든 응답에는 요구 티켓이 담깁니다. 이 결정이 요구하는 챌린지(알고리즘, 작업 비트, 개방 수)와 규칙의 해시를, 이 클라이언트 전용으로 도출한 키 아래 키 기반 BLAKE3 태그로 봉인한 것이며, 챌린지 수명 한 번 동안 유효합니다.
2. 중간 페이지는 `GET /__sibuna/challenge.json?path=<original URL>&need=<ticket>`을 가져옵니다. 유효한 티켓이 있으면 티켓이 챌린지를 결정합니다. 작업량은 그것을 요구한 결정으로 고정되며, 보고된 URL과 이 가져오기 요청 자체의 헤더로 다시 계산하지 않습니다. 이 헤더(`Accept`, `Sec-Fetch-*`)는 탐색 요청의 헤더와 다르기 때문입니다. 유효한 티켓이 없으면(URL을 무시하는 API 클라이언트, 수명이 지나도록 열어 둔 페이지) 서버는 보고된 URL을 평가합니다. URL은 요청 줄을 읽는 것과 같은 함수로 경로와 쿼리로 나누고, 경로는 요청 허용이 쓰는 정규형으로 바꿉니다(SID 0009). 8 KiB보다 긴 URL은 잘라 내지 않고 `414`로 거부하며, 정규형이 없는 경로는 `400`으로 거부합니다. 티켓은 참고용일 뿐입니다. 요청 허용은 여전히 세션의 작업 수준을 확인하므로, 재전송하거나 위조한 티켓은 그 보유자에게 작업 비용만 지울 수 있습니다.
3. 응답은 챌린지 레코드입니다.

   ```
   {"id":"AQEND...70
   chars","algorithm":"posw","difficulty":13,"challenges":16,"expires_at":1757241234}
   ```

   `hashcash`에서는 `difficulty`가 비트 수이고 `challenges`는 `0`입니다.
4. Web Worker가 WebAssembly 모듈로, WebAssembly를 쓸 수 없으면 JavaScript 증명자로 (제VII부) 이를 풀고, 다음 둘 중 하나를

   ```
   {"challenge_id":"...","nonce":"90766"}
   ```

   또는

   ```
   {"challenge_id":"...","proof":"<base64url, 32(1 + t(n+1))바이트>"}
   ```

   `POST /__sibuna/verify`로 보냅니다.
5. 성공하면 서버는 `Set-Cookie` 헤더와 함께 `200 {"status":"ok"}`로 응답하고, 페이지는 원래 위치를 다시 불러옵니다. 이후의 모든 요청은 쿠키를 함께 보내며 정책 평가 전에 허용됩니다.

경로가 챌린지와 함께 전달되는 이유 중간 페이지는 보호된 URL에서 제공되므로 브라우저는 자신이 어디에 있는지 압니다. 경로를 보내면 서버가 규칙별 난이도와 알고리즘을 고를 수 있고(결제 경로는 작업 비트 20의 순차 작업을, 블로그는 Hashcash 16비트를 요구할 수 있습니다), 규칙 해시를 토큰에 묶을 수 있습니다. 토큰의 규칙 해시는 `X-Sibuna-Rule-Hash`로 업스트림에 보고됩니다.

상태 없는 챌린지 레코드

학습 목표 챌린지 인코더를 읽고, 각 필드의 목적을 이해하며, 핫 패스에서 엔트로피 시스템 호출 없이 챌린지별 nonce를 도출하는 방법을 살펴봅니다.

구현: `Coordinator::create_challenge_with_spec` 36바이트 페이로드를 만들고 챌린지 키로 태그를 붙인 뒤, 52바이트를 base64url로 인코딩해 70문자 식별자를 만듭니다. `crates/challenge/src/coordinator.rs`(crate `sibuna-challenge`)에 있습니다.

```
pub fn create_challenge_with_spec(
    &self,
    client_ip: &[u8],
    user_agent: &[u8],
    now: u64,
    spec: ChallengeSpec,
    rule_hash: u64,
) -> Challenge {
    self.adaptive.observe(now.saturating_mul(1000));
    let effective = ChallengeSpec {
        difficulty: spec.difficulty.saturating_add(self.adaptive.bump()),
        ..spec
    };
    let fingerprint = self.fingerprint(client_ip, user_agent);
    let difficulty = effective.parameter();
    let challenges = match effective.algorithm {
        Algorithm::Hashcash => 0,
        Algorithm::Posw => effective.posw_challenges,
    };

    let mut payload = [0u8; ID_PAYLOAD_LEN];
    payload[0] = ID_VERSION;
    payload[1] = effective.algorithm.to_byte();
    payload[2] = difficulty;
    payload[3] = challenges;
    payload[4..12].copy_from_slice(&now.to_le_bytes());
    payload[12..20].copy_from_slice(&fingerprint.to_le_bytes());
    payload[20..28].copy_from_slice(&self.next_nonce(now, fingerprint).to_le_bytes());
    payload[28..36].copy_from_slice(&rule_hash.to_le_bytes());
    let mut raw = [0u8; ID_RAW_LEN];
    raw[..ID_PAYLOAD_LEN].copy_from_slice(&payload);
    raw[ID_PAYLOAD_LEN..].copy_from_slice(&self.challenge_tag(&payload));

    Challenge {
        id: encoded::encode(&raw),
        algorithm: effective.algorithm,
        difficulty,
        challenges,
        expires_at: now.saturating_add(u64::from(self.challenge_ttl)),
    }
}
```

식별자는 Encoded<70>입니다. 텍스트는 정확히 그 길이의 배열에 담기므로 챌린지를 발급할 때 할당이 일어나지 않습니다. ChallengeSpec::parameter는 작업 비트를 계층 고유의 단위로 바꿉니다. Hashcash 비트(최대 64) 또는 PoSW 깊이(difficulty - 3, 4에서 24 사이로 제한)입니다.
세 가지 세부 사항을 눈여겨볼 만합니다.

- 적응형 난이도. 코디네이터는 챌린지 발급률의 지수 가중 이동 평균을 원자 변수 세 개에 유지합니다. 기준선인 초당 챌린지 50개를 넘으면 발급률이 두 배가 될 때마다 작업 비트가 하나씩 늘며, 최대 여섯 비트까지 늘어납니다. 폭주 트래픽은 지수적으로 더 많은 비용을 치르고, 조용한 트래픽은 기본 비용을 유지합니다.
- *nonce*는 *PRF* 출력입니다. 챌린지 키를 키로 하여 카운터, 시계, 지문 위에서 계산합니다. 카운터 덕분에 유일하고 키 덕분에 예측할 수 없으므로, 공격자는 발급받지 않은 식별자에 대한 풀이를 미리 계산할 수 없습니다. 또한 핫 패스는 엔트로피를 얻으려고 운영체제를 호출하지 않습니다.
- 난이도는 태그 안에 담겨 전달됩니다. 클라이언트는 레코드를 고쳐 난이도를 낮출 수 없습니다. 그렇게 하면 증명을 살펴보기도 전에 태그 검사가 실패합니다.

검증 순서와 일회 사용

학습 목표 값싼 검사를 증명보다 먼저 실행하는 이유, 증명이 유효하다고 확인된 뒤에야 챌린지를 사용 완료로 표시하는 이유, 그리고 한 풀이를 동시에 두 번 제출할 때 어떻게 해결되는지 이해합니다.

구현: Coordinator::verify_and_mint 식별자를 디코딩하고 인증한 뒤 경과 시간과 클라이언트 바인딩을 확인하고, 기록된 계층에 맞게 증명을 검증하고, 태그를 사용 완료로 기록한 다음 세션 토큰을 발급합니다. crates/challenge/src/coordinator.rs(crate sibuna-challenge)에 있습니다.

```
pub fn verify_and_mint(
    &self,
    challenge_id: &[u8],
    solution: Solution<'_>,
    client_ip: &[u8],
    user_agent: &[u8],
    now: u64,
) -> Result<Verified, VerifyError> {
    let record = self.decode(challenge_id)?;
    let expires_at = record.issued_at.saturating_add(u64::from(self.challenge_ttl));
    if now > expires_at || record.issued_at > now.saturating_add(CLOCK_SKEW_SECONDS) {
        return Err(VerifyError::ChallengeExpired);
    }
    if record.fingerprint != self.fingerprint(client_ip, user_agent) {
        return Err(VerifyError::FingerprintMismatch);
    }
    check_solution(challenge_id, &record, solution)?;
    self.spent.mark_spent(&record.tag, expires_at, now).map_err(|error| match error {
        SpendError::DoubleSpend => VerifyError::DoubleSpendAttempt,
```

```
        SpendError::Full => VerifyError::StoreFull,
    })?;
    let work = WorkLevel {
        algorithm: record.algorithm.to_byte(),
        bits: record.algorithm.work_bits(record.difficulty),
    };
    Ok(Verified {
        token: self.mint_token(now, record.rule_hash, record.fingerprint, work),
        algorithm: record.algorithm,
        difficulty: record.difficulty,
        challenges: record.challenges,
        ttl_seconds: self.token_ttl,
    })
}
```

이 순서는 의도적입니다. 태그, 경과 시간, 바인딩은 상수 시간 검사로, 해시를 계산하기 전에 쓰레기 입력을 거부합니다. 틀린 증명이 챌린지를 소모해서는 안 되므로, 사용 완료 집합에는 `check_solution`이 성공한 뒤에만 씁니다. 같은 유효한 풀이를 검증한 두 스레드는 `mark_spent`에서 경쟁합니다. 샤드 락이 둘을 직렬화하므로 정확히 하나만 이기고, 다른 하나는 `VerifyError::DoubleSpendAttempt`를 받습니다. `issued_at > now + 60` 조건(`CLOCK_SKEW_SECONDS`는 60)은 시계가 1분 넘게 앞선 노드가 만든 레코드를 거부합니다. 그렇지 않으면 그런 레코드가 챌린지의 수명을 늘릴 수 있습니다.

오류는 Elm 스타일 진단(제X부)과 함께 `400`으로 클라이언트에 전달됩니다. 각 `VerifyError`는 `sibuna_core::errors::ErrorKind`로 변환되며, 그 설명이 응답 본문이 됩니다. 종단 간 테스트는 응답 본문의 `DOUBLE SPEND`, `FINGERPRINT MISMATCH`, `WRONG SOLUTION TYPE`, `MALFORMED CHALLENGE`를 확인합니다.

세션 쿠키

학습 목표 키 기반 해시 토큰을 읽고, 쿠키 속성을 이해하며, 클라이언트 바인딩을 설명합니다.

구현: `token::mac::mint`와 `token::mac::verify` 40바이트 페이로드를 직렬화하고, 그 위에서 계산한 키 기반 BLAKE3의 처음 16바이트를 덧붙이며, 상수 시간 비교로 검증합니다. `crates/challenge/src/token/mac.rs`(crate `sibuna-challenge`)에 있습니다. `Coordinator::verify_cookie`가 클라이언트 지문과 함께 이를 호출합니다.

```
pub fn verify(
    key: &[u8; 32],
    text: &[u8],
    now: u64,
    expected_fingerprint: Option<u64>,
) -> Result<Payload, TokenError> {
    if text.len() != TOKEN_LEN {
        return Err(TokenError::InvalidTokenLength);
    }
```

```
    let raw: [u8; RAW_LEN] =
        encoded::decode::<RAW_LEN, TOKEN_LEN>(text).ok_or(TokenError::InvalidEncoding)?;
    let (payload, given) = raw.split_at(PAYLOAD_LEN);
    let mut bytes = [0u8; PAYLOAD_LEN];
    bytes.copy_from_slice(payload);
    // 상수 시간: 중간에 빠져나가면 위조자가 태그의 몇 바이트가 맞았는지 알게 됩니다.
    if !ct::eq(&tag(key, &bytes), given) {
        return Err(TokenError::InvalidTokenSignature);
    }
    let payload = Payload::from_bytes(&bytes)?;
    payload.check(now, expected_fingerprint)?;
    Ok(payload)
}
```

태그는 페이로드를 파싱하기 전에 확인합니다. 다른 버전의 페이로드는 진본임이 확인된 뒤에야 `TokenError::InvalidEncoding`으로 거부되며, 위조된 페이로드는 파서에 도달하지 못합니다.
쿠키는 다음과 같이 내보냅니다.

```
Set-Cookie: __sibuna_token=<75자>; Path=/; Max-Age=86400; HttpOnly;
    SameSite=Lax[; Secure]
```

`HttpOnly`는 페이지 스크립트가 쿠키를 읽지 못하게 하고, `SameSite=Lax`는 최상위 탐색은 허용하면서 교차 사이트 재전송을 막습니다. TLS가 데몬 앞에서 종료되면 `--secure-cookie`로 `Secure`를 추가합니다. 페이로드에 주소와 User-Agent의 키 기반 지문이 담기므로, 다른 기계로 복사한 쿠키는 `TokenError::TokenBoundAddressMismatch`로 실패합니다. 종단 간 테스트 "the cookie is bound to the client identity"가 바로 이 경로를 시험합니다.
페이로드는 버전 바이트와, 보유자가 치른 작업 수준으로 시작합니다. 작업 수준은 방식(Hashcash 또는 PoSW)과 실제로 푼 작업 비트이며, 부하에 따른 추가 비트도 포함합니다. 세션은 그 수준이 경로가 요구하는 수준에 도달할 때에만 챌린지를 통과합니다. 경로의 요구 수준은 규칙의 난이도에서 챌린지 자체와 같은 제한 방식으로 계산합니다. 수준에는 이름이 아니라 순서가 있습니다. 24비트로 얻은 쿠키는 모든 16비트 경로에도 통하지만, 16비트 쿠키를 24비트 경로에 제시하면 더 강한 중간 페이지로 넘어가고 새 쿠키가 옛 쿠키를 대체합니다. `rule_hash`는 챌린지를 발급한 규칙의 감사용 식별자로 남습니다. 업스트림에 보고될 뿐 요청 허용에는 쓰이지 않습니다. WAF 탐지 결과와 명시적 거부는 세션이 있어도 결코 해제되지 않습니다. 종단 간 테스트 "a session earned on a cheaper route does not admit a route that demands more work"가 두 방향을 모두 시험합니다.

전달된 주소 인그레스 뒤에서는 클라이언트 주소가 `X-Forwarded-For`로 도착합니다. Sibuna는 `--trust-forwarded`가 있을 때에만 이 헤더를 신뢰합니다(forward-auth 모드에서는 자동으로 적용됩니다). 인터넷에 직접 노출된 리버스 프록시에서는 이 옵션을 꺼 두어야 합니다. 그렇지 않으면 어떤 클라이언트든 자신의 바인딩과 속도 제한 버킷을 스스로 고를 수 있습니다.

연습 문제 4.1. 같은 NAT 주소 뒤에서 User-Agent가 똑같은 두 브라우저의 순서도를 그려 보세요. 두 브라우저가 공유하는 검사와 공유하지 않는 검사는 무엇이며, 그 NAT에 있는 공격자가 다른 브라우저의 쿠키를 재사용하려면 무엇이 필요할까요?

불변식 설명하기. 데몬이 아직 풀리지 않은 챌린지를 "기억"하지 않는 이유와, 이것이 지름길이 아니라 보안 기능인 이유를 운영자에게 설명하세요.

제V부

Rust 무할당 엔지니어링

고정 버퍼 하나를 따라 연결을 *accept*부터 응답까지 추적하고, 여러 연결 스레드가 요청 경로에서 힙 할당 없이 상태를 공유하게 해 주는 동시성 데이터 구조를 살펴봅니다. 메모리 안전성을 지키려면 별도의 장치(읽는 쪽 카운트, 아레나, 디스크립터 테이블, 해제 연쇄)가 필요했을 곳에서는 *Rust* 소유권이 그 역할을 대신하는 방식을 보여 줍니다.

연결 루프

학습 목표 이 장을 마치면 64 KiB 버퍼 하나가 keep-alive 연결 전체를 처리하는 방식, 헤드와 본문을 복사 없이 찾는 방식, 모든 연결이 개수가 제한된 전용 스레드를 갖는 이유, 프록시가 원본 서버의 프레이밍을 알아내 프록시된 클라이언트 연결을 열어 둘 수 있는 방식, 그리고 이 보장 가운데 빌림 검사기가 대신 증명해 주는 것이 무엇인지 설명할 수 있어야 합니다.

연결당 스레드 하나, 개수는 제한

accept 워커(`--workers`, 기본값은 CPU당 하나, 최대 64)는 리스닝 소켓을 공유합니다. 받아들인 연결은 각각 1메가바이트 스택을 가진 전용 스레드에 넘기고, accept 루프는 곧바로 `accept`로 돌아갑니다. 연결 처리를 마친 스레드는 대기하다가 다음 연결을 맡습니다. 연결마다 스레드를 만들고 끝내면 다른 스레드가 멈췄기 때문입니다(제VIII부, SID 0002). 다른 방식, 즉 accept 스레드에서 연결을 끝까지 처리하는 방식이 첫 설계였습니다. 이 방식은 클라이언트 하나로는 측정 결과가 좋았지만 64개로는 나빴습니다. 그중 60개가 워커가 요청 256개 할당량을 마칠 때까지 기다렸고, 기다리는 클라이언트가 늘 때마다 꼬리 지연이 커졌기 때문입니다. 연결당 스레드를 쓰면 연결 64개에서 p99가 모든 전체 제품 워크로드에서 305 us에서 857 us 사이로, 1밀리초 미만에 머뭅니다(`benchmarks/results/product-latest.md`, 제VIII부).

연결 스레드 수는 `--max-connections`(기본값 1,024, 최대 8,192)로 제한됩니다. 한도를 넘으면 accept 루프는 아무것도 생성하지 않고 새 소켓에 `503 Service Unavailable`로 응답한 뒤 소켓을 닫고, 이를 `sibuna_overloaded_total`에 집계합니다.

구현: `accept_loop` 연결을 받아들이고, 각 연결을 연결 한도와 대조한 뒤, 작업 풀에서 전용 스레드를 받아 실행합니다. `cmd/sibuna/src/server.rs`(crate `sibuna`)에 있습니다.

```
fn accept_loop(server: &Arc<Server>, listener: &TcpListener) {
    while !server.is_stopping() {
        let stream = match listener.accept() {
            Ok((stream, _)) => stream,
            // ... 디스크립터나 메모리가 부족하면 스핀하지 않고 1 ms 물러난다
```

```
        };
        let limit = server.max_connections.load(Ordering::Relaxed);
        if server.connections.fetch_add(1, Ordering::Relaxed) >= limit {
            server.connections.fetch_sub(1, Ordering::Relaxed);
            server.metrics.overloaded.bump();
            let text = "Service Unavailable: connection limit reached";
            reject(server, &stream, text);
            continue;
        }
        set_no_delay(&stream);
        let task_server = Arc::clone(server);
        let launched = server.tasks.launch(move || {
            let _counted = Counted(&task_server);
            crate::connection::serve(&task_server, &stream);
        });
        // ... 실행에 실패하면 자리를 돌려주고 과부하로 집계한다
    }
}
```

Counted는 가드이며, 그 Drop은 스레드가 반환으로 끝나든 패닉으로 끝나든 연결이 차지한 한도의 자리를 돌려줍니다. 503 경로는 소켓에서 자기 쪽을 종료하기 전에 클라이언트가 이미 보낸 데이터를 비워 냅니다. 읽지 않은 바이트가 남은 소켓을 닫으면 리셋으로 응답하게 되고, 그 리셋이 503을 버리기 때문입니다.

스레드는 join할 수 있는 슬롯으로 이루어진 고정 테이블인 TaskPool(cmd/sibuna/src/tasks.rs)에서 가져옵니다. 끝난 스레드는 join될 때까지 스택을 유지하므로 슬롯은 다시 차지될 때 join되며, 새 작업은 커서를 전진시키지 않고 끝난 슬롯 가운데 가장 낮은 슬롯을 가져갑니다. 커서를 쓰면 테이블을 한 바퀴 다 돌아야 끝난 슬롯에 다시 닿으므로 끝난 스레드 수천 개가 메모리에 남고, 메모리는 열린 연결 수가 아니라 처리한 연결 수에 따라 늘어납니다. 두 번째 가드인 Finished는 작업이 끝나면 슬롯을 빈 것으로 표시하므로, 패닉한 연결도 슬롯을 반납합니다.

메모리도 같은 방식으로 제한됩니다. 연결 스레드는 64 KiB 읽기 버퍼와 16 KiB 쓰기 버퍼를 스택에, 40 KiB 프록시 작업 공간을 힙에 두며, 힙 공간은 연결이 시작될 때 한 번만 할당합니다. 따라서 최악의 경우 메모리 사용량은 클라이언트가 열 수 있는 소켓 수에 좌우되지 않고 미리 알 수 있는 값입니다. 유휴 연결 회수기(아래 참조)는 보내기를 멈춘 연결을 닫으므로, 느린 클라이언트는 --idle-timeout을 넘어 스레드를 붙잡을 수 없습니다.

할당자 선택

Rust 1.100은 Allocator API의 핵심과 Vec, “Box“의 할당자 매개변수를 안정화합니다(릴리스 노트). 우리는 프로세스의 할당자를 교체하지 않고 개별 컬렉션의 할당자를 선택할 수 있습니다. 이 API로 요청 경로의 할당을 더 줄일 수는 없습니다. 준비된 연결의 요청은 이미 메모리를 할당하지 않으며, “cmd/sibuna/tests/steady_state.rs“와 “crates/net/tests/steady_state.rs“가 이를 검사합니다. 프록시는 연결마다 세 버퍼를 한 번 할당하고 재사용합니다. 제한기와 사용된 증명 집합도 테이블을 한 번만 할당합니다. 0으로 초기화하는 할당 덕분에 건드리지 않은 페이지는 상주하지 않아도 되며, “crates/store/tests/resident.rs“가 이를 검사합니다.

할당자 변경을 조사할 때 우리는 먼저 정책 스냅샷 구성과 콘솔 JSON 파싱을 측정합니다. 임시 컬렉션은 요청 경로 밖에서 메모리를 할당하지만 컬렉션의 할당자가 요소의 할당까지 자동으로 관리하지는 않습니다. JSON 트리는 이스케이프된 문자열도 “Cow<str>“를 통해 소유하며, 소유 형태는 “String“입니다. Rust 1.100의 “String“에는 할당자 매개변수가 없습니다.

“Arc::new_in“도 이 릴리스에서는 여전히 불안정합니다(API 문서). 게시한 엔진은 마지막 읽기 사용자가 해제할 때까지 “Arc“로 공유되므로 재구성 중에 이전 엔진이 아직 사용하는 메모리를 초기화할 수 없습니다. 아레나는 이러한 수명과 파서의 입력 한도를 보존해야 합니다.
우리는 “cmd/console-ui/src/abi.rs“에 있는 콘솔 WebAssembly 모듈의 기존 전역 할당자 “WipeOnFree“를 유지합니다. 이 할당자는 재할당 후의 이전 블록을 포함하여 해제하는 블록을 지웁니다. 컬렉션에 “System“을 직접 선택하면 이 지우기를 우회합니다. 작업 증명 WebAssembly 모듈에는 힙 할당자가 없고, Argon2도 호출자가 소유하는 작업 공간을 이미 재사용합니다. 따라서 측정한 작업 부하가 변경을 정당화할 때까지 기존 할당자와 Rust 1.94 호환성을 유지합니다. 후보는 지우기, 한도가 있는 입력 처리, 할당 없는 요청 경로를 보존하면서 지연 시간이나 상주 메모리를 개선해야 합니다.

연결당 버퍼 하나

요청 경로의 모든 동적 할당은 서비스 거부 공격의 지렛대가 됩니다. 며칠에 걸친 단편화, 스레드 사이의 할당자 락 경합, 그리고 할당이 최대가 되도록 요청을 조작하는 공격자의 증폭이 그렇습니다. Sibuna의 규칙은 절대적입니다. 분류, 검증, 프록시 전달은 힙 할당 없이 동작합니다. 연결 처리기는 소켓의 수명 동안 64 KiB 읽기 버퍼와 16 KiB 쓰기 버퍼를 소유하며, 그 밖의 모든 것은 이 버퍼를 가리키는 슬라이스입니다.

구현: `connection::serve` 연결을 유휴 연결 회수기에 등록하고, 버퍼, 프록시 작업 공간, 엔진 캐시를 한 번 준비한 뒤 요청을 최대 4,096개 처리합니다. `cmd/sibuna/src/connection.rs`(crate `sibuna`)에 있습니다.

```
pub fn serve(server: &Server, stream: &TcpStream) {
    let Some(lease) = server.idle.register(stream) else { return };
    // ...
    let mut read_buffer = [0u8; MAX_REQUEST_BYTES]; // 64 KiB
    let mut write_buffer = [0u8; WRITE_BUFFER]; // 16 KiB
    let mut reader = Reader::new(stream, &mut read_buffer[..]);
    let mut peer = PeerText::default();
    peer.format(stream.peer_addr().ok());
    let mut link = Link {
        server,
        stream,
        writer: Writer::new(stream, &mut write_buffer[..]),
        workspace: Workspace::new(),
        cache: EngineCache::new(),
        activity: lease.activity(),
        last: false,
    };
    for served in 0..MAX_REQUESTS_PER_CONNECTION {
        link.last = served + 1 == MAX_REQUESTS_PER_CONNECTION;
        if !serve_one(&mut reader, &mut link, peer.as_bytes()) {
            break;
        }
        lease.touch();
    }
}
```

read_head는 빈 줄이 나타날 때까지 버퍼를 채우되 마지막 탐색 이후 도착한 바이트만 탐색하며, 16 KiB를 넘는 헤드는 431로 거부합니다. parse_request(crate sibuna-http)는 메서드, 경로, 쿼리, 헤더, 쿠키가 모두 버퍼의 슬라이스인 Request<'a>를 만듭니다. 선언된 본문은 들어갈 수 있는 만큼 채운 뒤 헤드 뒤에서 슬라이스로 잘라 냅니다. 그다음 reader.consume(head_len + body_len)이 리더를 전진시키므로 다음 keep-alive 요청은 깨끗하게 시작합니다. 리더는 공간이 필요할 때에만 읽지 않은 바이트를 앞으로 옮깁니다. chunked 본문은 헤드 뒤에서 제자리에서 디코딩합니다(SID 0009). 각 단계의 프레이밍을 버퍼에서 잘라 내므로 디코딩된 앞부분은 연속된 상태로 남아 검사할 수 있습니다. 내부 경로는 길이로 구분되며 연결을 열어 둡니다. 프록시된 요청도 원본 서버의 응답에 프레이밍이 있고 본문을 모두 버퍼에 담았다면 연결을 열어 둡니다. 프록시가 이를 어떻게 결정하는지는 다음 절에서 봅니다.

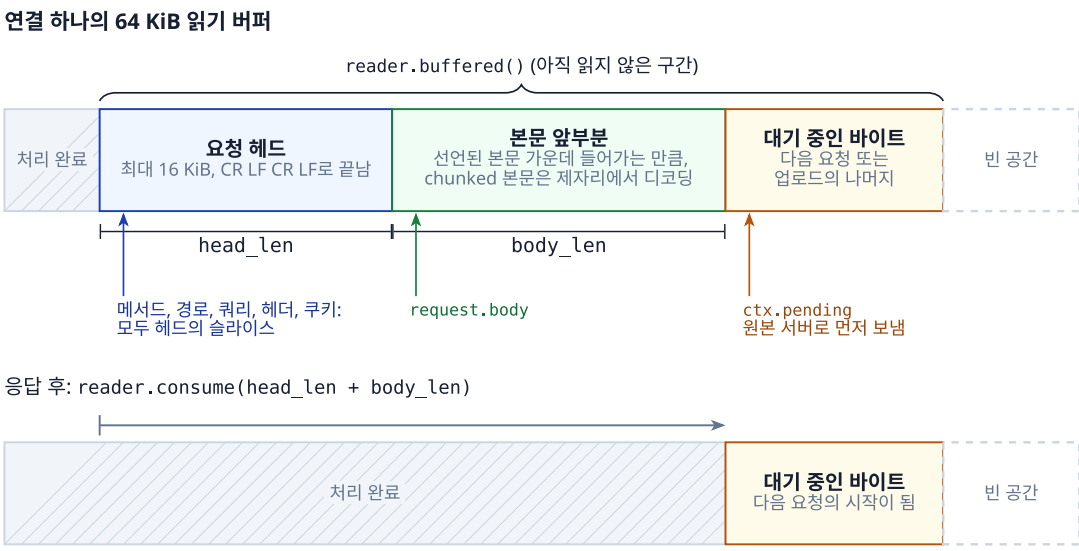


그림 6 요청 하나를 처리하는 동안의 연결 읽기 버퍼. 요청의 모든 필드는 이 버퍼의 슬라이스입니다.

```
fn serve_one<W: Write>(
    reader: &mut Reader<'_, &TcpStream>,
    link: &mut Link<'_, W>,
    peer: &[u8],
) -> bool {
    let head_len = match read_head(reader) {
        Head::Complete(length) => length,
        Head::TooLarge => { /* 431 "Request head exceeds 16 KB" */ return false; }
        Head::Closed | Head::Failed => return false,
    };
    let framing = {
        let bytes = reader.buffered();
        let head = bytes.get(..head_len).unwrap_or_default();
        let request = match parse_request(head) {
            Ok(request) => request,
            Err(error) => return reject_head(link, error),
        };
        let declared = request.content_length().unwrap_or(0);
        // ... Expect: 100-continue
```

```
        if declared == 0 && !request.chunked {
            let upload = Upload::declared(&request);
            let pending = bytes.get(head_len..).unwrap_or_default();
            let keep = dispatch(link, request, pending, upload, 0, peer);
            reader.consume(head_len);
            return keep;
        }
        (declared, request.chunked)
    };
    serve_with_body(reader, link, peer, head_len, framing)
}
```

빌림 검사기가 증명하는 것 요청 슬라이스는 다음에 버퍼를 채울 때까지만 유효합니다. Sibuna는 이 규칙을 주석이 아니라 타입으로 나타냅니다. 요청은 `Request<'a>`이고, `'a`는 리더 버퍼에 대한 공유 빌림입니다. 버퍼를 채우거나 소비하려면 `&mut reader`가 필요한데, 컴파일러는 그 버퍼에서 빌린 `Request`가 아직 쓰이는 동안에는 이를 거부합니다. `serve_one`에서 요청은 `dispatch`로 이동하고, 그 호출이 반환된 뒤에야 `reader.consume`을 실행할 수 있습니다. 순서를 거꾸로 쓰면 컴파일되지 않습니다.

같은 규칙이 처음에는 이상해 보이는 두 가지 선택을 설명합니다. 첫째, `serve_with_body`는 헤드를 두 번 파싱합니다. chunked 본문을 버퍼에 담으면 버퍼 안의 바이트가 움직이므로, 첫 파싱은 본문을 버퍼에 담기 전에 빌림을 끝내야 합니다. 두 번째 파싱은 더는 바뀌지 않는 버퍼를 빌립니다. 파싱은 결정적이며 쿠키가 있는 브라우저 요청에 약 133 ns가 걸리므로 (`benchmarks/results/primitives-latest.md`의 `zero_copy_request_and_cookie`, 132.5 ns), 두 번째 파싱은 비용이 거의 들지 않고 같은 판정을 냅니다. 둘째, 프록시는 파싱된 요청과 대기 중인 바이트를 빌리며 연결 버퍼를 다시 채워서는 안 되므로, 클라이언트가 더 보내는 바이트는 전용 업로드 버퍼로 읽습니다. 읽은 것을 되돌려 줄 수 없으므로, 클라이언트 연결은 본문을 모두 버퍼에 담았을 때에만 유지됩니다.

프록시 헤드 재작성

헤드는 그대로 전달하지 않습니다. `write_request_head`(`crates/net/src/request.rs`, crate `sibuna-net`)는 요청 줄과 각 헤더를 원본 라이터로 다시 내보내면서 다음을 제거합니다. 홉 단위 필드(`Connection`, `Keep-Alive`, `Proxy-Connection`, `Transfer-Encoding`, `TE`, `Trailer`, `Upgrade`, 프록시 인가 필드), 요청에 이미 들어 있는 전달 필드나 감사 필드(`X-Forwarded-For`, `X-Real-IP`, `Forwarded`, `X-Forwarded-Proto`, `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Sibuna-Status`, `X-Sibuna-Rule`), `Expect`, `Content-Length`, 그리고 요청의 `Connection` 헤더가 지정하는 모든 필드입니다. 그런 다음 감사 필드 집합을 덧붙입니다. 본문 프레이밍은 복사하지 않고 새로 만듭니다.

```
for header in request.headers() {
    if is_hop_by_hop(header.name) || nominated(request, header.name) {
        continue;
    }
    out.write_all(header.name)?;
    out.write_all(b": ")?;
    out.write_all(header.value)?;
    out.write_all(b"\r\n")?;
```

```
}
match upload {
    Upload::None => {}
    Upload::Length(length) => {
        out.write_all(b"Content-Length: ")?;
        write_decimal(out, *length)?;
        out.write_all(b"\r\n")?;
    }
    Upload::Chunked(_) => out.write_all(b"Transfer-Encoding: chunked\r\n")?,
}
```

Content-Length는 다시 내보냅니다. 연결 버퍼 안에서 끝난 chunked 본문은 제자리에서 디코딩해 길이와 함께 보냅니다. 더 긴 본문은 Transfer-Encoding: chunked로 알리고 읽기 한 번마다 다시 청크로 나눕니다(SID 0009). 읽은 데이터는 프록시의 16 KiB 업로드 버퍼에 들어가 그 자리에서 디코딩되고 정규화된 청크 하나로 나가므로, 원본 서버는 클라이언트의 청크 크기, 확장, 트레일러를 결코 보지 못합니다. 감사 필드 집합은 다음과 같습니다.

```
Connection: keep-alive
X-Forwarded-For: <client>
X-Forwarded-Proto: <scheme>
X-Real-IP: <client>
X-Sibuna-Status: PASS
X-Sibuna-Rule: session | robots-txt | ip/cidr-trie | ...
```

버퍼보다 큰 본문은 응답을 스트리밍해 돌려주기 전에 클라이언트에서 원본 라이터로 16 KiB 조각 단위로 릴레이합니다(relay_body). crates/net/src/request.rs의 단위 테스트 proxy_head_rewrite_drops_hop_by_hop_headers_and_injects_audit_fields는 재작성이 위조된 X-Forwarded-For를 제거하고 감사 필드를 주입하는지 확인합니다.

프록시의 버퍼 세 개, 즉 원본 읽기 버퍼(응답 헤드 하나, 16 KiB), 원본 쓰기 버퍼(8 KiB), 업로드 버퍼(16 KiB)는 연결이 한 번 할당해 모든 요청에 재사용하는 Workspace 안에 있습니다. 작업 공간 덕분에 버퍼의 수명은 교환 하나가 아니라 연결 전체와 같아지고, 원본 리더와 라이터는 한 번에 교환 하나 동안만 버퍼를 빌립니다.

원본 응답 릴레이

첫 프록시는 원본 서버가 연결을 닫을 때까지 원본의 바이트를 스트리밍한 뒤 클라이언트 연결을 닫았습니다. 파싱 없이 올바른 프레이밍을 얻지만, 프록시하는 요청마다 새 TCP 연결이라는 대가를 치렀습니다. 부하 생성기를 돌리면 이 대가가 TIME_WAIT 상태의 소켓 수만 개로, 루프백에서는 임시 포트 고갈로 드러났습니다. 이제 프록시는 원본의 헤드(최대 16 KiB)를 읽고 RFC 9112의 규칙에 따라 본문을 분류합니다.

> **구현: parse_response_head** 상태, 네 가지 프레이밍 중 하나(HEAD 요청과 1xx, 204, 304 응답에는 본문 없음; Content-Length; chunked 전송 코딩; 연결 종료로 구분), 그리고 원본 서버가 연결을 유지하는지를 반환합니다. crates/net/src/response.rs(crate sibuna-net)에 있습니다.

```
pub enum Framing { None, Length(u64), Chunked, UntilClose }
```

```
pub fn parse_response_head(
    head: &[u8],
    head_request: bool,
) -> Option<ResponseHead> {
    let status = status_code(head)?;
    let mut fields = FramingFields {
        keep_alive: head.get(7) == Some(&b'1'),
        ..Default::default()
    };
    for line in crlf_lines(head).skip(1) {
        if line.is_empty() {
            break;
        }
        fields.read_line(line)?; // Transfer-Encoding, Content-Length, Connection
    }
    let mut framing = fields.length.map_or(Framing::UntilClose, Framing::Length);
    if head_request || status / 100 == 1 || status == 204 || status == 304 {
        framing = Framing::None;
    } else if fields.encoded {
        framing =
            if fields.chunked { Framing::Chunked } else { Framing::UntilClose };
    }
    let mut keep_alive = fields.keep_alive;
    if framing == Framing::UntilClose || fields.closing {
        keep_alive = false;
    }
    if fields.nominates_framing {
        return None;
    }
    Some(ResponseHead { status, framing, keep_alive })
}
```

파서는 프록시가 릴레이해서는 안 되는 것을 거부합니다. 잘못된 상태 줄, 토큰이 아닌 필드 이름, 값 안의 제어 바이트, 서로 충돌하는 길이, 반복되거나 마지막이 아닌 chunked 코딩, Content-Length나 Transfer-Encoding을 지정하는 Connection 필드입니다. 파서의 작업량은 헤드 길이에 선형이며, 결코 패닉하지 않습니다.

헤드는 원본의 Connection 헤더를 Sibuna 자신의 결정으로 바꾼 뒤 클라이언트에 다시 내보내고, 본문은 정확히 릴레이합니다. 길이가 있으면 copy_exact, chunked 코딩이면 청크 단위(크기 줄, 데이터, 트레일러), 레거시 방식이면 copy_remaining을 씁니다. 클라이언트 연결을 닫는 것은 레거시 방식뿐입니다. 모든 릴레이는 하나의 복사 단계 위에 만들어져 있습니다. 소스가 이미 가진 데이터는 모두 전달하고, 아무것도 없을 때에만 대상을 플러시한 뒤 한 번 읽습니다. 따라서 작은 쓰기가 이어지는 스트림은 만들어지는 즉시 전달되고, 대량 전송은 여전히 읽기와 쓰기 한 번마다 버퍼 하나를 가득 옮깁니다.

```
let keep = client_keep_alive && parsed.framing != Framing::UntilClose;
write_client_head(client, head, keep).map_err(client_head_error)?;
origin.consume(head_length);
match parsed.framing {
    Framing::None => {}
    Framing::Length(length) => copy_exact(origin, client, length, progress)?,
    Framing::Chunked => relay_chunked(origin, client, progress)?,
```

```
    Framing::UntilClose => copy_remaining(origin, client, progress)?,
}
client.flush().map_err(|_| ProxyError::ClientFailed)?;
Ok(Relayed { client_keep: keep, origin_reusable: parsed.keep_alive })
```

필드 여덟 개짜리 원본 헤드를 파싱하는 데 322 ns가 걸리고, 응답이 1 KiB인 keep-alive GET을 루프백으로 프록시하는 데 29.8 us가 걸립니다(2026년 10월 측정, crates/net/src/lib.rs에 측정 방법이 문서화되어 있습니다).

원본 연결 풀

프레이밍을 파싱하면 원본 소켓을 다시 쓸 수 있는지도 알 수 있습니다. Connection: close가 없는 HTTP/1.1 응답(또는 keep-alive가 있는 HTTP/1.0 응답)의 본문을 모두 소비했다면, 소켓은 깨끗한 요청 경계에 놓입니다. 그런 소켓은 유휴 원본 연결 256개짜리 고정 풀에 들어가고, 다음에 프록시하는 요청은 새로 연결하는 대신 그 소켓을 가져갑니다. 풀이 없으면 프록시하는 요청마다 새 원본 연결을 열고, 닫힌 연결마다 로컬 포트를 TIME_WAIT에 붙잡아 둡니다. 부하가 걸리면 루프백 위의 프록시는 곧 임시 포트가 바닥나고 처리량이 급락합니다.

풀을 락 하나로 보호하면 프록시하는 모든 요청이 그 락을 두고 경합합니다. 그래서 풀(crates/net/src/pool.rs)은 스트라이프로 나뉩니다. 소켓 16개짜리 스트라이프 16개이며, 각 스트라이프는 각자의 캐시 라인에 놓인 Mutex<Vec<Arc<TcpStream>>>입니다. 스레드는 처음 쓸 때 라운드 로빈으로 홈 스트라이프를 배정받고, 소켓을 반납하거나 가져갈 때 그 스트라이프를 먼저 쓰며, 새로 연결하기 전에 다른 스트라이프도 살펴봅니다. 이때 다른 스레드가 잡은 락은 기다리지 않습니다.

```
pub fn take(&self) -> Option<Arc<TcpStream>> {
    if self.is_stopping() {
        return None;
    }
    let home = home_stripe();
    if let Some(socket) =
        self.stripe(home).and_then(|stripe| lock(&stripe.sockets).pop())
    {
        return Some(socket);
    }
    (1..STRIPES).find_map(|offset| {
        let stripe = self.stripe((home + offset) % STRIPES)?;
        try_lock(&stripe.sockets)?.pop()
    })
}
```

풀에 있던 소켓은 유휴 상태로 있는 동안 원본 서버가 이미 닫았을 수 있습니다. 프록시는 이를 두 가지 방식, 즉 쓰기 실패나 첫 응답 바이트 전의 스트림 끝으로 알아차립니다. 두 경우 모두 아직 클라이언트에 아무것도 전달되지 않았으므로, 소켓을 닫고 새 연결로 한 번 재시도합니다. 다른 풀 소켓으로는 결코 재시도하지 않습니다. 유휴 기간이 지나면 풀의 소켓이 모두 이미 끊겼을 수 있는데, 두 번째 풀 소켓을 가져가던 이 재시도의 첫 버전은 조용한 기간이 지날 때마다 첫 요청에 502로 응답했습니다. 재시도는 본문을 모두 버퍼에 담은 안전한 메서드(GET, HEAD, OPTIONS)에만 허용됩니다. 응답이 없다고 원본 서버가 변경을 커밋하지 않았다는 증거가 되지는

않으며, 조각으로 릴레이한 본문은 다시 보낼 수 없습니다. 두 경우 모두 클라이언트는 502를 받습니다.

```
let safe = matches!(request.method, Method::Get | Method::Head | Method::Options);
let buffered = exchange.upload.is_buffered(request.body.len());
let retryable = safe && buffered;
for attempt in 0..2 {
    // 재시도는 언제나 새로 연결한다. 유휴 기간이 지나면 풀의 소켓이 모두
    // 끊겼을 수 있고, 끊긴 소켓을 또 쓰면 요청이 실패한다.
    let pooled = if attempt == 0 { pool.take() } else { None };
    // ... 풀에 소켓이 없으면 연결하고, activity에 붙인 뒤 교환한다
    match outcome {
        Ok(relayed) => {
            if relayed.origin_reusable {
                pool.give(origin);
            }
            return Ok(relayed.client_keep && buffered);
        }
        Err(error) => {
            drop(origin);
            let stale = was_pooled
                && retryable
                && matches!(
                    error,
                    ProxyError::UpstreamClosed | ProxyError::UpstreamWriteFailed
                );
            // ... 그 밖의 오류는 즉시 보고한다
        }
    }
}
```

단위 테스트는 고정된 바이트 문자열을 같은 함수로 릴레이하고, 연결 헤더를 제외하면 출력이 바이트 단위로 같은지 확인합니다(crates/net/src/relay/tests.rs). 데몬의 종단 간 테스트는 클라이언트 소켓 하나로 프록시 요청 두 개를 보내고, 스텁 원본 서버가 붙이는 순번 헤더로 두 번째 요청이 풀의 원본 연결을 재사용했는지 확인합니다. crates/net/tests/의 벡터 테스트는 기록된 프록시 시나리오(원시 클라이언트 요청과 미리 준비한 원본 응답)를 재생하고 원본 서버와 클라이언트가 받는 모든 바이트를 비교합니다.

> **원본 소켓이 Arc인 이유** 소켓 디스크립터는 작은 정수이며, 커널은 디스크립터가 닫히자마자 재사용합니다. 종료는 진행 중인 교환을 중단시켜야 하지만, 그사이 다른 연결에 재사용된 디스크립터를 건드려서는 안 됩니다. 락으로 보호하는 활성 디스크립터 테이블로 이를 막을 수 있겠지만, 그러면 교환마다 락 비용이 듭니다. Sibuna에는 그런 테이블이 필요 없습니다. 원본 소켓은 Arc<TcpStream>입니다. 풀이 핸들 하나를, 교환이 하나를 들고, 교환이 실행되는 동안 연결의 Activity가 복제본 하나를 듭니다. 디스크립터는 마지막 핸들이 버려질 때 닫히므로, 누군가 아직 접근할 수 있는 동안에는 닫히거나 재사용될 수 없습니다.

유휴 기한

모든 클라이언트 연결은 소켓을 IdleTable(crates/net/src/idle.rs)에 등록하고 Lease를 받습니다. 리스의 Activity는 연결에서 바이트가 마지막으로 움직인 시각을 담습니다. 데몬은 요청을 하나 처

리할 때마다 이를 갱신하고, 프록시는 원본에서 읽을 때마다, 그리고 업로드 16 KiB마다 갱신하며 진행 중인 교환의 원본 소켓을 붙여 둡니다. 회수기 스레드는 가장 짧은 타임아웃의 4분의 1마다(200 ms에서 1초 사이) 깨어나, 타임아웃보다 오래 조용했던 모든 연결을 붙어 있는 원본 소켓과 함께 종료합니다. 그러면 어느 쪽에서든 막혀 있던 스레드가 스트림 끝을 보고 슬롯을 놓아줍니다.

따라서 기한은 응답이 얼마나 긴지가 아니라 연결이 얼마나 오래 조용했는지를 잽니다. 계속 움직이는 큰 다운로드는 결코 끊기지 않습니다. 업로드 바이트는 신뢰할 수 없는 쪽에서 오므로 16 KiB마다 한 번만 시각을 갱신합니다. 이는 클라이언트에게 유휴 기간마다 최소 16 KiB를 요구하는 셈으로, 최소 전송 속도 기반 요청 타임아웃이 제공하는 느린 본문 공격 방어와 같습니다.

테이블에는 슬롯마다 작은 락이 하나 있고 테이블 전체 락은 없습니다. 위에서 말한 이유로 테이블은 각 클라이언트 소켓의 `try_clone`과 붙어 있는 각 원본 소켓의 `Arc`를 담습니다. `Lease`는 `Drop`에서 연결 등록을 해제하며, 회수된 슬롯은 그때까지 리스가 소유하므로, 옛 연결이 새 연결이 차지한 슬롯을 해제하는 일은 결코 일어나지 않습니다.

업그레이드된 연결

검증된 `101` 뒤에는 WebSocket 업그레이드를 양방향으로 릴레이합니다. 각 방향은 고정 버퍼 하나를 가진 전용 스레드에서 실행됩니다. 연결 자신의 스레드는 원본에서 클라이언트로 가는 바이트를, 128 KiB 스택을 가진 scoped 스레드는 클라이언트에서 원본으로 가는 바이트를 나릅니다. 스레드 하나가 `poll(2)`로 두 방향을 모두 처리할 수도 있지만 Rust 표준 라이브러리는 이를 노출하지 않습니다. 소켓 타임아웃을 건 스레드 두 개를 쓰면 릴레이에 `unsafe`가 필요 없습니다. 따라서 업그레이드된 연결은 스레드 두 개를 차지하므로, 모든 연결이 업그레이드되면 연결 스레드 수가 `--max-connections`의 두 배에 이를 수 있습니다.

두 방향은 유휴 기한 하나(`--websocket-idle-timeout`, 기본값 300초)를 공유합니다. 어느 방향으로 바이트가 움직이든 같은 시각 하나를 갱신합니다. 소켓 대기마다 남은 시간과 같은 타임아웃을 두며, 타임아웃이 난 대기는 공유 시각을 다시 읽고 나머지 시간만큼 다시 기다리거나 릴레이를 끝냅니다. 한 방향이 스트림 끝에 이르면 다른 소켓의 보내는 쪽을 종료합니다(half close). 한 방향이 실패하면 두 소켓을 모두 종료해 다른 방향도 끝나게 합니다. 업그레이드를 프록시하기 전에 연결은 캐시한 정책 엔진을 놓아주므로, 몇 시간 동안 살아 있을 수 있는 연결이 옛 엔진을 살려 두지 않습니다.

중지

`Running::stop`과 `Running`의 `Drop`은 모두 정해진 순서로 데몬을 중지합니다. 중지 플래그를 설정하고 리스너에 연결해 모든 accept 워커를 깨우고, 워커를 join하고, 등록된 모든 클라이언트 소켓과 붙어 있는 원본 소켓을 종료하고(`IdleTable::shutdown`), 원본 연결 풀을 멈추고, 모든 연결 스레드를 join하고, 회수기를 join한 뒤, 풀의 소켓을 닫습니다.

SIGINT와 SIGTERM을 받으면 평범한 Rust 코드를 거쳐 이 절차가 실행됩니다. 데몬은 스레드를 띄우기 전에 두 시그널을 블록하고(이후의 모든 스레드는 이 마스크를 물려받습니다) 스레드 하나를 `sigwait` 전용으로 둡니다. 이 스레드는 요청을 기록하고 메인 스레드를 즉시 깨웁니다. 시그널 처리기 안에서 실행되는 코드는 없고, 플래그를 폴링하는 스레드도 없습니다. 시그널 모듈(`cmd/sibuna/src/signals.rs`)에는 데몬의 유일한 `unsafe` 코드가 있습니다. `std`가 감싸지 않는 몇 가지 C 라이브러리 함수의 선언입니다.

Drop으로 해제하기 명시적인 정리 호출은 준비의 역순으로 써야 하고, 모든 조기 반환을 그 호출과 대조해 읽어야 합니다. Sibuna에서는 각 자원이 Drop에서 스스로를 해제합니다. Daemon과 Running은 서버를 멈추고, Lease는 연결 등록을 해제하며, Counted는 연결 한도의 자리를 돌려주고, Finished는 작업 슬롯을 놓아줍니다. 컴파일러는 범위를 빠져나가는 모든 경로에 각 해제를 삽입하며, 범위를 거쳐 되감기(unwinding)되는 패닉도 예외가 아닙니다. 따라서 어떤 경로도 해제를 잊을 수 없습니다.

측정한 내용

표는 Ryzen 9 5950X 한 대에서 리버스 프록시 모드로 실행한 Shield를 보여 줍니다. 데몬은 CPU 네 개에 고정하고, wrk는 다른 CPU 여덟 개에서 스레드 두 개와 연결 64개로 실행했으며, 5초씩 세 번 실행한 중앙값입니다. 행은 benchmarks/results/product-latest.md에서 옮겨 왔으며, 전체 기록은 제VIII부에 있습니다.

표 7 리버스 프록시 모드의 Shield(출처: benchmarks/results/product-latest.md)

워크로드	상태	req/s	p50 / p99 (us)	요청당 CPU 시간(us)
세션으로 허용, 프록시됨	200	138,965	438 / 857	28.57
챌린지(중간 페이지)	200	195,188	162 / 328	13.73
정적 경로 허용, 프록시됨	200	145,673	415 / 746	27.27
WAF 거부(세션 있음)	403	209,507	151 / 306	12.17

wrk는 요청 64개를 계속 진행 중으로 유지하므로, CPU가 병목인 프록시 경로에서는 요청이 CPU 네 개를 기다리며 줄을 서고, 여기서 p99는 요청 하나의 비용이 아니라 스케줄러의 꼬리입니다. 반면 정해진 속도로 보내고 각 요청을 원래 보내려던 시각부터 재는 개방 루프 생성기는 같은 제공 부하에서 지연을 측정합니다. 리버스 프록시 모드에서 허용된 요청의 p99는 초당 요청 10,000개에서 65 us, 100,000개에서 100 us입니다. 150,000개를 제공하면 CPU 네 개가 포화되어 초당 134,344개 요청을 처리하고, 큐 대기는 수 초로 늘어납니다(benchmarks/results/latency-latest.md). 예열된 keep-alive 연결은 모든 결정을 힙 할당 없이 처리합니다. cmd/sibuna/tests/steady_state.rs는 데몬의 할당을 세며, crates/net/tests/steady_state.rs는 프록시한 keep-alive 요청 500개 동안 할당을 하나도 찾지 못합니다.

연습 문제 5.1. 클라이언트가 200 KB를 업로드합니다. 어떤 바이트가 64 KiB 버퍼에 머무는지, 어떤 바이트를 relay_body가 업로드 버퍼를 거쳐 릴레이하는지, 그리고 프록시가 릴레이하기 전에 헤드를 원본 서버에 써야 하는 이유를 추적하세요.

할당 없는 동시성 상태

학습 목표 샤드 락, Robin Hood 사용 완료 집합, GCRA 제한기, 버전이 있는 차단 테이블, MPSC 인시던트 링, 그리고 `Arc`와 세대 확인 캐시를 이용한 정책 엔진의 read-copy-update 게시를 분석합니다.

`sibuna-store`의 모든 저장소는 만들어질 때 고정 용량을 한 번 할당하며, 그 뒤로는 할당하지도 전역 락을 기다리며 멈추지도 않습니다. 백그라운드 청소기가 필요한 저장소도 없습니다. 만료된 상태는 그 자리에서 회수됩니다.

락과 샤딩

Sibuna 테이블의 임계 구역은 명령어 수십 개 길이로, 커널 futex 왕복보다 훨씬 짧습니다. 테이블은 키 해시로 샤드 16개에 나뉘므로 두 연산이 경합할 확률은 1/16이며, 각 샤드는 각자의 캐시 라인에 놓이므로 이웃한 락끼리 거짓 공유가 일어나지 않습니다.

테이블은 `std::sync::Mutex`를 사용합니다. 경합이 없으면 compare-and-swap 한 번, 보유자가 실행 중이면 짧은 스핀, 그다음 futex 대기입니다. 대안은 대기 루프에 스핀 루프 힌트를 넣은 두 상태 원자적 스핀 락입니다. SID 0010은 둘 사이에서 측정으로 고르도록 요구했습니다. 스핀 락은 경합 없는 연산마다 약 1나노초를 아끼지만, 경합이 생기면 뒤처집니다.

표 8 Ryzen 9 5950X에서의 샤드 락 처리량(2026년 10월)

워크로드	`Mutex`	스핀 락
사용 완료 집합, 스레드 16개	10.8 M/s	8.6 M/s
사용 완료 집합, 스레드 64개 (CPU 수보다 많음)	11.2 M/s	6.9 M/s
요청이 몰리는 주소 여섯 개에 대한 GCRA, 스레드 16개	44.9 M/s	19.3 M/s
요청이 몰리는 주소 여섯 개에 대한 GCRA, 스레드 64개	37.4 M/s	6.5 M/s

엣지 서버는 CPU 수보다 많은 연결 스레드를 실행하며, 보유자가 선점된 스핀 락은 모든 대기자가 자기 시간 조각을 태우게 만듭니다. 그래서 뮤텍스를 유지합니다. 오염된(poisoned) 락은 오류를 전파하지 않고 그대로 진입합니다. 어떤 연산도 샤드를 잡은 동안 패닉하지 않으며, 설령 패닉하더라도 샤드는 구조적으로 유효한 상태로 남습니다.

Robin Hood 사용 완료 집합

사용 완료 집합에는 받아들여진 증명만 들어갑니다(제III부). 집합에는 샤드 16개가 있고, 각 샤드는 `Option<Entry>` 슬롯 4,096개짜리 개방 주소 배열이며, 항목은 `{ tag: [u8; 16], expires_at: u64, dist: u8 }`입니다. Robin Hood 삽입(Celis, 1986)은 들어오는 키보다 자기 홈에 더 가까운 점유자를 밀어내므로 탐사 길이의 편차가 작게 유지됩니다. 조회는 찾는 키가 자기 홈에서 떨어진 거리보다 자기 홈에 더 가까이 있는 항목을 만나는 즉시 멈춥니다.

구현: `Table::insert` 사용 완료 태그를 만료 시각과 함께 삽입하며, 지나가는 길에 만료된 슬롯을 회수합니다. `crates/store/src/spent.rs`(crate `sibuna-store`)에 있습니다.

```
fn insert(
    &mut self,
    tag: &Tag,
    expires_at: u64,
    now: u64,
) -> Result<(), SpendError> {
    if !self.can_insert(tag, now) {
        return Err(SpendError::Full);
    }
    let mut carry = Entry { tag: *tag, expires_at, dist: 0 };
    let mut idx = Self::home(tag);
    while carry.dist < MAX_PROBE {
        let slot = &mut self.entries[idx];
        let Some(e) = slot else {
            *slot = Some(carry);
            return Ok(());
        };
        // 만료된 점유자를 덮어써도 안전한 것은 새 거리가 더 작지 않을 때뿐이다.
        // 그래야 이 슬롯을 지나 탐사하는 키가 여전히 조기 종료 검사를
        // 통과한다.
        if e.is_expired(now) && carry.dist >= e.dist {
            *e = carry;
            return Ok(());
        }
        if carry.dist > e.dist {
            std::mem::swap(&mut carry, e);
            // 밀려난 만료 항목은 그냥 버린다.
            if carry.is_expired(now) {
                return Ok(());
            }
        }
        idx = (idx + 1) % SHARD_CAPACITY;
        carry.dist += 1;
    }
    Err(SpendError::Full)
}
```

미묘한 부분은 만료된 슬롯 규칙입니다. 만료된 항목은 자기 홈에서 적어도 그만큼 먼 키로만 제자리에서 덮어쓸 수 있습니다. 그렇지 않으면 그 슬롯을 지나 탐사하는 나중 키가 일찍 멈춰 사라집니다. 밀려난 만료 항목은 그냥 버립니다. 청소 스레드는 필요 없습니다. 삽입은 먼저 아무것도 바꾸지 않고 밀어내기 경로를 모의 실행하여(`can_insert`), 횟수가 제한된 밀어내기가 살아 있는 태그를 잃지 않고 성공할 수 있는지 확인합니다.

불변식 거부된 삽입은 살아 있는 태그를 결코 쫓아내지 않습니다. 따라서 사용되었고 아직 만료되지 않은 태그는 언제나 찾을 수 있으며, 거부(`SpendError::Full`)는 서비스를 거부할 수는 있어도 재전송을 가능하게 하지는 않습니다.

슬롯은 `Option<Entry>`입니다. 빈 슬롯에는 읽을 태그가 없으므로, 읽는 쪽이 확인을 잊을 수 있는 `occupied` 플래그가 없고, `match`는 두 경우를 모두 처리할 때까지 컴파일되지 않습니다. 오류 타입도 마찬가지로 정확합니다. 사용 완료 집합은 두 가지 방식(`DoubleSpend`와 `Full`)으로 실패할 수 있으며, `SpendError`는 정확히 그 둘만 나타냅니다.

GCRA: 정수 하나로 하는 속도 제한

Generic Cell Rate Algorithm(ATM Forum, 1996)은 누출 버킷의 가상 스케줄링 형태입니다. 클라이언트마다 이론적 도착 시각(TAT) 하나를 저장합니다. 윈도 W당 양의 한도 N에 대해 방출 간격 $T = \max(1, \lceil W/N \rceil)$, 버스트 허용치 $\tau = (N-1)T$로 두면 다음과 같습니다.

$$\text{arrival at } t \ \text{ conforms} \iff \mathrm{TAT} \le t + \tau, \quad \text{then} \ \ \mathrm{TAT} \leftarrow \max(\mathrm{TAT}, t) + T.$$

정리(토큰 버킷 상한) t_1에서 첫 적합 도착이 일어난 뒤, k번째 적합 도착은 $t_k - t_1 \ge (k-1)T - \tau$를 만족합니다. 따라서 길이 L인 어떤 구간에도 적합 도착은 최대 $N + \lfloor L/T \rfloor$개입니다. N개의 버스트 뒤에 T마다 하나입니다. 정의된 버스트를 넘어서는 고정 윈도 경계의 부작용이 없고, 요청별 이력도 없습니다.

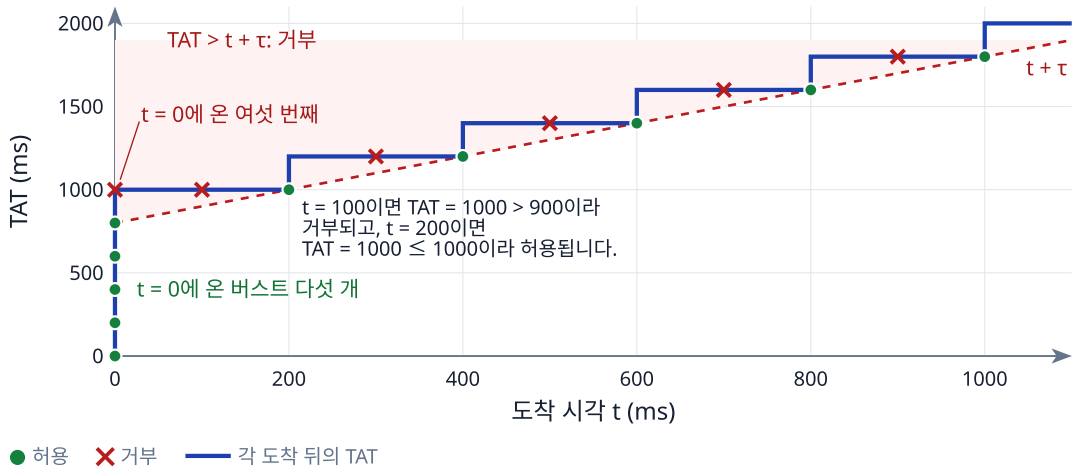


그림 7 초당 5개 한도의 GCRA 허용: 버스트 다섯 개 뒤에 200 ms마다 하나. 도착 시점의 TAT가 $t + \tau$ 선에 걸치거나 그 아래에 있으면 그 도착은 허용됩니다.

풀이 예제: 초당 다섯 개를 손으로 계산하기 $N = 5$, $W = 1000$ ms로 두면 $T = 200$ ms, $\tau = 800$ ms입니다. 유휴 상태였던 클라이언트가 $t = 0$에 요청 여섯 개를 보낸다고 합시다. 첫 요청은 빈 셀을 만납니다. $\mathrm{TAT} = \max(0, 0) = 0 \le 800$이므로 적합하며 $\mathrm{TAT} \leftarrow 200$이 됩니다. 다음 네 요청도 같은 방식으로 적합하며 $\mathrm{TAT} = 1000$을 남깁니다. 여섯 번째 요청은 $1000 > 0 + 800$을 만나 `Retry-After` $\mathrm{TAT} - \tau - t = 200$ ms와 함께 거부됩니다. $t = 199$의 요청은 여전히 $1000 > 999$를 만나 거부되고, $t = 200$의 요청은 $1000 \le 1000$을 만나 적합하며 $\mathrm{TAT} = 1200$을 남깁니다. `crates/store/src/limiter.rs`의 단위 테스트 `gcra_admits_exactly_the_burst_then_drains_at_the_emission_rate`가 바로 이 숫자들을 확인합니다.

구현: arrive 락을 잡은 샤드의 셀에 도착 하나를 적용하고 결정, 재시도 지연, 남은 예산을 반환합니다. crates/store/src/limiter.rs(crate sibuna-store)에 있습니다.

```
fn arrive(
    cells: &mut [Cell; SLOTS_PER_SHARD],
    key: u64,
    now_ms: u64,
    limits: Limits,
) -> Decision {
    let interval = limits.emission_interval();
    let tau = limits.burst_tolerance();
    let Some(cell) = locate(cells, key, now_ms, tau) else {
        return Decision {
            limited: true,
            capacity_exhausted: true,
            retry_after_ms: interval,
            remaining: 0,
        };
    };
    let tat = cell.tat_ms.max(now_ms);
    let horizon = now_ms.saturating_add(tau);
    if tat > horizon {
        return Decision {
            limited: true,
            capacity_exhausted: false,
            retry_after_ms: tat - tau - now_ms,
            remaining: 0,
        };
    }
    cell.tat_ms = tat.saturating_add(interval);
    // ... remaining = min(rate, floor((t + tau - TAT) / T) + 1)
}
```

샤드의 check는 락을 잡고, 락으로 보호된 셀을 넘겨 arrive를 호출합니다. check가 반환되면 가드가 락을 풉니다. 셀은 16바이트(키와 TAT)이며, 16슬롯 탐사 윈도을 가진 512슬롯 샤드 16개에 있어 모두 합쳐 약 128 KiB입니다. TAT가 $t - \tau$보다 오래된 셀은 이미 다 빠진 상태이므로 그 자리에서 회수됩니다. Retry-After 헤더도 같은 산술로 계산합니다. 탐사 윈도이 가득 차면 활성 할당량 상태를 쫓아내는 대신 새 클라이언트를 거부하며, 결정이 그 사실을 알립니다(capacity_exhausted). 그래서 호출자는 포화로 인한 거부를 차단으로 격상하지 않습니다. 설정된 속도가 0이면 요청을 거부합니다. 제한기 하나가 여러 규칙을 처리합니다. 셀 키는 규칙의 범위를 시드로 한 클라이언트 주소의 Wyhash이므로, 서로 다른 범위의 같은 클라이언트는 독립된 예산을 가집니다.

버전이 있는 차단 테이블

차단에는 원자적 버전, 키, 만료 필드를 가진 슬롯 4,096개를 쓰며, 차단된 주소의 키 기반 해시로 8슬롯 탐사 윈도의 개방 주소 방식을 씁니다. 쓰는 쪽은 뮤텍스로 다른 쓰는 쪽과 직렬화되며, 갱신마다 앞뒤로 버전을 증가시킵니다. 읽는 쪽은 락을 잡지 않고, 안정된 짝수 버전만 받아들이며, 동시 변경이 있으면 재시도합니다.

구현: `BanTable::ban`과 `BanTable::read_slot` 쓰는 쪽이 갱신을 버전 증가로 감싸는 방식과 읽는 쪽의 일관된 스냅숏입니다. `crates/store/src/ban_table.rs`(crate `sibuna-store`)에 있습니다.

```
pub fn ban(&self, banned: &[u8], until: u64, now: u64) {
    let key = slot_key(banned);
    let _writers = self.lock_writers();
    let victim = &self.slots[self.victim(key, now)];
    // 키와 만료 시각은 논리적으로 한 값이다. 교체하는 동안 버전을 홀수로
    // 두어, 읽는 쪽이 옛 키와 다음 키의 만료 시각을 섞지 못하게 한다.
    victim.version.fetch_add(1, Ordering::SeqCst);
    victim.until.store(until, Ordering::SeqCst);
    victim.key.store(key, Ordering::SeqCst);
    victim.version.fetch_add(1, Ordering::SeqCst);
}

fn read_slot(&self, slot: &Slot) -> (u64, u64) {
    loop {
        let before = slot.version.load(Ordering::SeqCst);
        if before & 1 != 0 {
            std::hint::spin_loop();
            continue;
        }
        let key = slot.key.load(Ordering::SeqCst);
        let until = slot.until.load(Ordering::SeqCst);
        if before == slot.version.load(Ordering::SeqCst) {
            return (key, until);
        }
    }
}
```

순차 일관성은 옛 식별자와 새 만료 시각이 섞이는 것을 막습니다. 모든 접근이 하나의 전체 순서에 속하므로 논증이 단순해집니다. 읽는 쪽은 뮤텍스를 잡지 않지만 쓰는 쪽을 기다릴 수 있으므로 wait-free는 아닙니다. 이 원자적 연산에는 실제 비용이 있으며, 공짜라고 가정하지 않고 HTTP 하니스에서 간접적으로 측정합니다. 탐사 윈도이 가득 차면 만료된 슬롯을, 없으면 가장 먼저 만료되는 슬롯을 덮어씁니다. `clear`는 모든 차단을 풀지만 키는 그대로 둡니다. 비운 슬롯은 탐사 연쇄에서 뒤에 있는 충돌 키를 숨길 수 있기 때문입니다.

MPSC 인시던트 링

Shield 운영 구성이 요청을 거부하거나 허니팟이 작동하면, 인시던트는 약 1.3 KB짜리 고정 레코드(클라이언트 주소, User-Agent, 메서드, 경로, 범주, 페이로드이며, 각각 용량에서 잘림)에 복사되어 크기가 제한된 다중 생산자 단일 소비자 링(Vyukov의 시퀀스 표식 설계, 슬롯 512개, `BoundedQueue<IncidentRecord, 512>`)에 들어갑니다. 생산자는 연결 스레드이고 소비자는 저장소 스레드입니다. 링이 가득 차면 응답을 막는 대신 가장 새로운 레코드를 버립니다. 이는 영속성 보장이 아니라 명시적인 손실 정책입니다. 버린 레코드를 세는 카운터로 그 손실을 관찰할 수 있습니다.

```
/// `incident`를 링에 복사한다. 링이 가득 차면 버리고 버린 횟수를 센다.
pub(crate) fn record(&self, incident: &Incident<'_>) {
```

```
    if self.queue.push(IncidentRecord::from_incident(incident)).is_err() {
        self.metrics.incidents_dropped.bump();
    }
}
```

레코드를 대기 중인 배치로 옮긴 뒤, 저장소 스레드는 커밋이 확인될 때까지 준비된 행과 함께 레코드를 보관합니다. 데이터베이스 오류가 나도 조용히 버리지 않으며, 재시도는 똑같은 배치를 씁니다.

각 슬롯에는 시퀀스 번호가 있어, 생산자에게는 슬롯이 비었는지를, 소비자에게는 슬롯이 찼는지를 알려 주므로 어느 쪽도 락을 잡지 않습니다. 이런 링은 안전한 Rust로 작성할 수 없습니다. 생산자와 소비자가 한 배열의 서로 다른 슬롯을 동시에 건드리며, 어떤 스레드가 어떤 슬롯을 건드려도 되는지는 타입 시스템이 아니라 시퀀스 프로토콜만이 증명하기 때문입니다. 그래서 crates/store/src/ring.rs 모듈은 값을 UnsafeCell<MaybeUninit<T>> 슬롯에 담고, 각 unsafe 블록은 자신을 정당화하는 프로토콜 단계를 밝힙니다.

```
if diff == 0 {
    // Relaxed: CAS는 어느 생산자가 위치 `pos`를 차지할지만 정한다.
    match self.tail.0.compare_exchange_weak(
        pos, pos.wrapping_add(1), Ordering::Relaxed, Ordering::Relaxed,
    ) {
        Ok(_) => {
            // SAFETY: CAS로 이 스레드가 위치 `pos`를 얻었고,
            // `sequence == pos`는 슬롯이 비었음을 뜻한다. 다른 생산자는
            // `pos`를 차지할 수 없고, 소비자는 아래의 Release 저장 전에는
            // 이 슬롯을 읽지 않는다.
            unsafe { (*slot.value.get()).write(value) };
            // Release: 쓴 값을 소비자에게 게시한다.
            slot.sequence.store(pos.wrapping_add(1), Ordering::Release);
            return Ok(());
        }
        Err(current) => pos = current,
    }
} else if diff < 0 {
    return Err(value);
}
```

링의 두 가지 규약은 주석이 아니라 타입입니다. 링이 가득 차면 push는 Err(value)를 반환해 레코드를 되돌려 주므로, 호출자가 셀지 버릴지 정합니다. false만 반환한다면 레코드를 잃습니다. 링에는 소비자가 하나뿐이며, pop은 &mut self를 받고, 공유 코드는 먼저 유일한 Consumer를 차지해야 하므로, 두 번째 소비자는 컴파일되지 않거나 실행 중에 거부됩니다.

Read-Copy-Update 엔진 게시

정책 엔진은 정책이 바뀔 때 핫 패스 밖에서 다시 만들어집니다. 요청은 절반만 만들어진 엔진을 결코 보아서는 안 되고, 요청이 아직 읽고 있는 엔진을 해제해서도 안 됩니다. Sibuna는 각 엔진을 EngineSlot 안의 Arc<Engine>으로 게시하며, 각 연결은 EngineCache를 통해 이를 읽습니다.

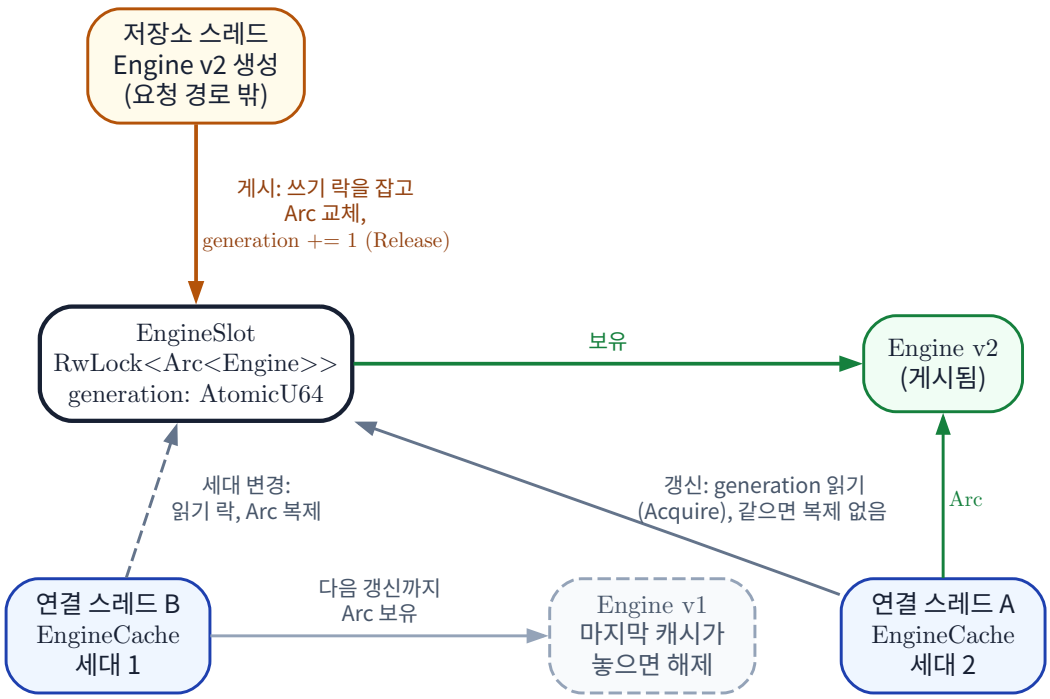


그림 8 엔진 게시. 저장소 스레드가 완성된 엔진을 게시하고 세대를 올립니다. 각 연결은 acquire 읽기 한 번으로 세대를 확인하고, 게시가 일어난 뒤에만 게시된 Arc를 복제합니다. 옛 엔진은 그 엔진을 가진 마지막 캐시가 놓아줄 때 해제됩니다.

구현: `EngineSlot::publish`와 `EngineCache::refresh` 완성된 엔진의 게시와, 연결 하나가 그 엔진을 보는 방식입니다. `cmd/sibuna/src/slot.rs`(crate `sibuna`)에 있습니다.

```rust
pub struct EngineSlot {
    current: RwLock<Arc<Engine>>,
    generation: AtomicU64,
}

pub fn publish(&self, engine: Engine) {
    let fresh = Arc::new(engine);
    let previous = {
        let mut current =
            self.current.write().unwrap_or_else(PoisonError::into_inner);
        std::mem::replace(&mut *current, fresh)
    };
    self.generation.fetch_add(1, Ordering::Release);
    // 옛 엔진은 락 밖에서 버린다(이것이 마지막 보유자였다면).
    drop(previous);
}

pub struct EngineCache {
    generation: u64,
    engine: Option<Arc<Engine>>,
```

```
}

pub fn refresh(&mut self, slot: &EngineSlot) -> &Engine {
    let generation = slot.generation();
    if self.generation != generation {
        // 위의 읽기와 이 복제 사이에 게시가 일어나면 새 엔진에 옛 세대 번호가
        // 붙는다. 그러면 다음 갱신이 다시 읽으면 된다.
        self.generation = generation;
        self.engine = None;
    }
    self.engine.get_or_insert_with(|| slot.load())
}
```

세대는 엔진이 자리 잡은 뒤 release 순서로 저장되고 acquire 순서로 읽힙니다. 새 세대를 본 읽는 쪽은 그 전에 저장된 엔진도 봅니다. 주석에 적힌 경쟁 상태는 무해합니다. 읽는 쪽이 세대를 읽은 뒤 복제하기 전에 게시가 일어나면, 읽는 쪽은 더 새로운 엔진을 더 오래된 세대 번호 아래에 캐시하게 되고, 다음 갱신 때 불일치를 알아차려 다시 읽습니다. 읽는 쪽은 요청 하나 동안 엔진 하나만큼 늦을 수는 있어도, 절반만 만들어진 엔진을 볼 수는 없습니다. 슬롯에는 완성된 엔진만 저장되기 때문입니다.

불변식 요청이 읽는 모든 엔진은 완성되어 있으며, 어떤 요청이나 연결 캐시가 엔진을 가지고 있는 동안에는 그 엔진이 해제되지 않습니다. 첫째는 `publish`가 저장하기 전에 엔진이 완전히 만들어지기 때문에 성립하고, 둘째는 모든 읽는 쪽이 `Arc`를 가지며 엔진은 마지막 `Arc`와 함께만 버려지기 때문에 성립합니다.

왜 그냥 락을 쓰지 않고 캐시를 쓸까요? SID 0010은 스레드 16개로 읽기 비용을 측정해 `RwLock<Arc<Engine>>`과 안전한 Rust로 만든 락 프리 대안 중에서 고르도록 요구했습니다. 락을 거쳐 읽으면 락 워드와 참조 카운트를 건드리는데, 이 둘은 모든 코어가 쓰기를 하는 캐시 라인 두 개입니다. 반면 세대는 게시할 때 한 번 쓰이고 그 밖에는 읽히기만 하므로, 각 코어가 자기 사본을 유지합니다.

표 9 엔진 읽기 한 번의 비용, Ryzen 9 5950X, 루프에서 읽는 스레드 16개(2026년 10월)

읽기 방식	스레드 1개	스레드 16개
`RwLock` 읽기 가드만	4.2 ns	752 ns
`RwLock` 읽기와 `Arc` 복제	7.6 ns	833 ns
캐시된 `Arc`로 세대 확인	0.8 ns	1.8 ns

경합 상황에서 캐시를 통한 읽기는 락보다 400배 넘게 저렴하므로, 요청 경로는 캐시를 쓰고 `RwLock`은 갱신의 느린 경로에서만 씁니다. 측정은 `cmd/sibuna/src/slot.rs`에 있는, 기본적으로 실행되지 않는(ignored) 테스트 `read_cost_under_contention`입니다. 요청 경로에서 쓸 때마다 갱신하므로, 요청은 각 단계 시점의 최신 엔진을 봅니다. 엔진 자체는 작습니다. 압축된 힙 테이블 209,052바이트, 약 204 KiB이며(`benchmarks/results/primitives-latest.json`의 `engine_bytes`), 규칙 128개, 128바이트 규칙 이름, 트라이 노드 8,192개를 담을 용량이 있습니다.

읽는 쪽 카운트가 아니라 `Arc`인 이유 고정 메모리 대안은 엔진 슬롯 두 개를 두며, 각 슬롯에는 읽는 쪽 카운트와 자기 할당 아레나가 있습니다. 요청은 카운트를 증가시키고 슬롯 포인터를 다시 확인해 슬롯을 고정합니다. 읽는 쪽이 읽은 뒤 증가시키기 전에 쓰는 쪽이 교체하는 경쟁을 막으려면 두 원자 연산이 모두 순차 일관적이어야 합니다. 빌더는 포인터를 교체한 다음, 옛 슬롯의 읽는 쪽이 모두 빠질 때까지 스핀한 뒤에야 그 슬롯의 아레나를 초기화하고 그 안에 다시 만듭니다. 따라서 재구성 시점에 걸쳐 슬롯을 붙잡고 있는 읽는 쪽은 빌더를 막으며, 모든 요청이 프록시 I/O를 시작하기 전에 슬롯을 놓지 않으면 느린 원본 서버가 다음 게시를 멈춰 세울 것입니다.
`Arc` 참조 카운팅에는 이런 것이 전혀 필요 없습니다. 저장소 스레드는 스냅숏마다 새 `Engine`을 만들어 게시합니다. 재사용하는 것이 없으므로 여분 슬롯도, 아레나 초기화도, 스핀도 없습니다. `publish`는 결코 읽는 쪽을 기다리지 않습니다. 옛 엔진은 마지막 보유자가 놓아줄 때까지 살아 있을 뿐이며, 느린 원본 서버는 그 순간만 늦출 수 있을 뿐 다음 게시는 늦추지 못합니다. 판정은 빌림을 끝내기 위해 일치한 규칙 이름을 128바이트 버퍼에 복사합니다. 결정은 연결의 캐시를 통해 엔진을 빌리고, 응답 경로는 연결 컨텍스트 전체를 가변으로 빌려야 하기 때문입니다. 몇 시간 동안 살아 있을 수 있는 업그레이드된 연결은 프록시되기 전에 캐시한 엔진을 놓아줍니다.

연습 문제 5.2. GCRA 정리를 써서, `--rate-limit 100 --rate-window 10`일 때 유휴 상태였던 클라이언트 하나가 처음 3초 동안 통과시킬 수 있는 요청의 최대 수를 계산하세요. 고정 10초 윈도 카운터, 그리고 엄밀한 슬라이딩 윈도 로그와 비교하세요.

불변식 설명하기. 만료된 Robin Hood 항목은 거리가 적어도 그만큼 큰 키로만 덮어쓸 수 있는 이유를 슬롯 세 개짜리 예로 설명하세요.

제V I부

오토마톤, 트라이, 의미 기반 방화벽

마이크로초 미만의 분류를 가능하게 하는 알고리즘을 살펴봅니다. 태그가 붙은 *Aho-Corasick* 오토마톤, *IPv4/IPv6* 기수 트라이, 의미 기반 *WAF*의 바이트 클래스 토크나이저, 그리고 *WEIGH* 점수를 쓰는 선언형 정책 엔진입니다.

한 번의 순회로 여러 패턴 매칭하기

학습 목표 이 장을 마치면 Aho-Corasick 오토마톤의 선형 시간 보장을 유도하고, 밀집 테이블 배치와 바이트 클래스를 통한 대소문자 접기를 설명하며, SIMD 리터럴 엔진을 채택하지 않은 이유를 논증할 수 있어야 합니다.

밀집 DFA로 구현한 Aho-Corasick

Sibuna에는 리터럴 패턴 집합이 두 개 있습니다. 봇 시그니처 약 40개와 공격 시그니처 약 100개입니다. 필드 하나를 패턴 N개와 하나씩 대조하면 $O(NM)$이 듭니다. Aho와 Corasick(1975)의 오토마톤은 실패 링크가 있는 트라이를 만든 뒤 필드를 $O(M)$에 한 번 훑습니다. Sibuna는 빌드할 때 실패 링크를 전이 테이블에 접어 넣으므로, 스캔은 입력 바이트마다 테이블 읽기 한 번이며 패턴 집합에 따른 분기가 없습니다. 오토마톤은 `sibuna-patterns` crate에 있습니다.

```
/// `state`(행 오프셋)에서 `byte` 하나만큼 전이합니다.
fn step(&self, state: usize, byte: u8) -> usize {
    self.table[state + usize::from(self.classes[usize::from(byte)])] as usize
}

pub fn find_first(&self, haystack: &[u8]) -> Option<Match> {
    let mut state = 0;
    for (index, &byte) in haystack.iter().enumerate() {
        state = self.step(state, byte);
        if state >= self.first_match {
            let id = self.first_output_in(self.match_state(state), self.all_tags)?;
            return Some(self.report(id, index + 1));
        }
    }
    None
}
```

상태는 평평한 테이블 하나 안의 행 오프셋으로 저장되므로, 한 단계는 덧셈 한 번과 읽기 한 번입니다. 상태 번호는 모든 매치 상태가 다른 모든 상태보다 뒤에 오도록 매깁니다. 그러면 매치 탐지는 `first_match`와의 비교 한 번으로 끝나며, 안쪽 루프에서 두 번째 테이블을 읽을 필요가 없습니다.

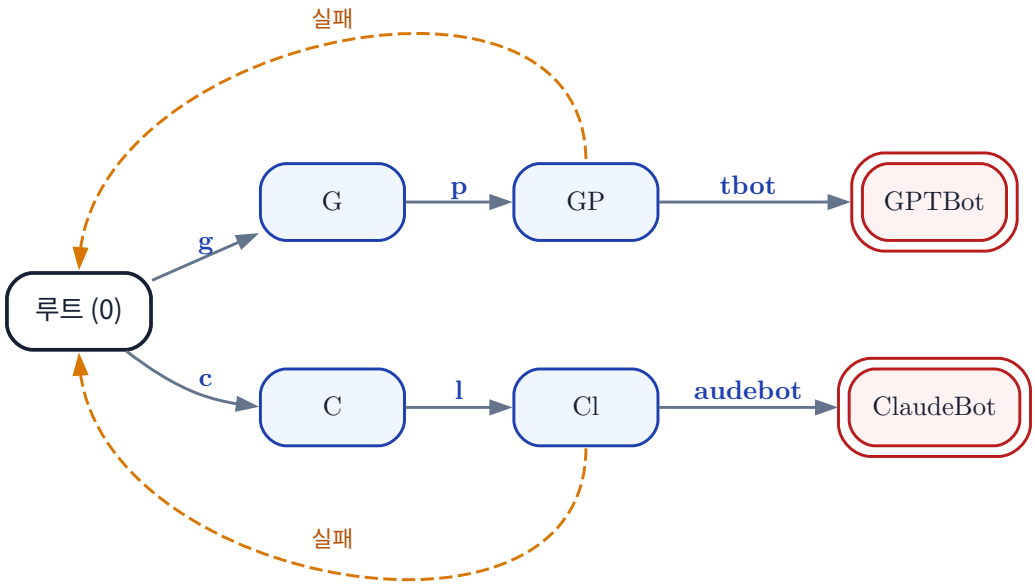


그림 9 두 시그니처의 트라이 상태와 실패 간선

오토마톤의 상태 예산은 실행 시점 옵션입니다. 정책 엔진은 루트 하나에 모든 시그니처 길이의 합을 더한 수만큼만 예약하며, 이는 트라이 상태 수의 안전한 상한입니다. 이로써 예전처럼 고정 테이블이 지나치게 커지는 일을 피하면서도 바이트 클래스마다 밀집 전이를 유지합니다. 각 패턴에는 작은 범주 태그(0에서 63까지이므로 태그 집합은 64비트 워드 하나)가 붙습니다. 대소문자 접기에는 오토마톤과 함께 만드는 256개 항목의 바이트 클래스 테이블을 사용합니다. 한 글자의 대문자와 소문자는 같은 클래스를 공유하고, 어떤 패턴에도 나오지 않는 바이트는 모두 클래스 하나를 공유합니다. 그래서 테이블 한 행의 너비는 256이 아니라 서로 다른 패턴 바이트 수에 1을 더한 값입니다. 엔진의 정확한 크기와 순차 부분 문자열 탐색과의 비교는 제VIII부에 기록되어 있습니다. 순차 탐색은 같은 패턴과 같은 any-match 의미론을 사용합니다.

정규 표현식, SIMD 엔진, 그리고 선택

시그니처 단계에 쓸 세 가지 계열을 비교했습니다.

- 백트래킹 정규 표현식(백트래킹 규칙 집합 방식)은 요청마다 수백 개의 패턴을 실행하고 매치마다 할당합니다. 가장 느린 선택지이며 오탐도 가장 많이 냅니다.
- *SIMD* 리터럴 엔진(Hyperscan, Vectorscan)은 셔플 기반 니블 마스크("Teddy" 알고리즘)로 16에서 32바이트 블록을 미리 걸러 내고 후보를 오토마톤으로 검증합니다. 큰 리터럴 집합으로 긴 입력을 다룰 때 뛰어납니다. 대가는 수 메가바이트짜리 C++ 의존성, 컴파일 시점 패턴 데이터베이스, WebAssembly 타깃 미지원이며, 100바이트 정도의 입력에서는 이점도 없습니다. 필드 전체를 스캔하는 시간이 캐시 미스 한 번보다 짧기 때문입니다.
- 밀집 *Aho-Corasick*과 작은 구조 토크나이저, 즉 libinjection 계보는 고정 테이블에서 정확한 선형 시간 동작을 제공하며, 바이너리 밖의 것은 아무것도 필요하지 않습니다.

Sibuna의 필드는 짧으므로 세 번째 계열이 메모리와 속도 모두에서 앞섭니다. 제VIII부는 8 KiB 본문에서 바이트당 약 1.56 ns를 기록합니다(의미 기반 스캔에 12,784.3 ns가 걸리며, `benchmarks/`

results/primitives-latest.md의 8kb_body_semantic_scan입니다). SIMD 사전 필터가 효과를 낼 만한 워크로드는 이것 하나뿐이며, SID 0006은 이를 미결 항목으로 남겨 둡니다.

IPv4와 IPv6를 위한 기수 트라이

> **학습 목표** 128비트 트라이에서 최장 접두사 조회를 추적하고, IPv4 루트 지름길을 설명하며, 이 트라이가 클러스터가 채우는 평판 저장소를 어떻게 겸하는지 살펴봅니다.

128비트 키 위의 이진 트라이 하나가 두 주소 계열을 모두 처리합니다. IPv6 접두사는 그대로 삽입하고, IPv4 접두사는 ::ffff:0:0/96으로 매핑하므로 /24는 깊이 120의 경로가 됩니다. 노드는 u32로 인덱싱하는 평평한 Vec 하나에 저장되며, 인덱스 0은 루트이자 "자식 없음"을 뜻합니다. 그래서 조회는 포인터 추적 없이 최대 128번의 의존 읽기입니다. 이 트라이는 sibuna-cidr crate의 PrefixTrie입니다.
IPv4 조회마다 매핑된 접두사의 96단계를 걸어 내려가면, IPv4 주소의 첫 비트에 닿기도 전에 의존 읽기 96번이 듭니다. 그래서 트라이는 초기화할 때 깊이 96의 노드를 기록해 두고 IPv4 조회를 거기서 시작합니다.

```
let (mut node, mut depth, mut best) = if is_ipv4_mapped(key) {
    let start = self.ipv4_root;
    let value = self.nodes[start as usize].value.as_ref();
    (start, IPV4_DEPTH, value.or(self.nodes[0].value.as_ref()))
} else {
    (0, 0, self.nodes[0].value.as_ref())
};
```

IPv4 주소 분류에는 32.0 ns, IPv6 주소 분류에는 79.3 ns가 걸립니다(benchmarks/results/primitives-latest.md의 ipv4_cidr_classification과 ipv6_cidr_classification). 엔진은 트라이를 두 번 참조합니다. 최종 판정인 allow/deny는 규칙 테이블보다 먼저 참조하므로 클러스터 전체의 차단이 모든 규칙에 우선하고, challenge 판정은 규칙 테이블 뒤에 참조합니다.

의미 기반 방화벽

> **학습 목표** 바이트 클래스 사전 스캔, 태그가 붙은 시그니처 오토마톤, 항진식 탐지를 갖춘 SQL 토크나이저, HTML과 셸 구조 검사, 정규화, 그리고 실제 브라우저 요청은 손대지 않고 통과시켜야 한다는 오탐 원칙을 이해합니다.

오탐의 교훈

WAF의 첫 버전은 모든 헤더를 평평한 부분 문자열 목록과 대조했습니다. 목록에는 /*가 있었고, 모든 브라우저는 Accept: text/html,...,*/*;q=0.8을 보냅니다. 그래서 실제 브라우저가 모두 SQL 인젝션으로 분류되어 403을 받았습니다. 이 교훈이 설계의 방향을 정했습니다. 탐지기는 구조에만 반응해야 합니다. 이제 테스트 모음은 select, --, 아포스트로피가 들어 있는 댓글 본문을 포함한 완전한 Chrome 요청을 엔진에 통과시키고, 아무것도 탐지되지 않는지 확인합니다.

한 번의 스캔으로 바이트 클래스 수집하기

검사하는 필드는 한 번 스캔해 어떤 바이트 클래스가 나타나는지 모읍니다. 따옴표, =, <, 셸 구분자, %, NUL, 정규화 트리거, 연속된 공백입니다. 모든 탐지기는 다시 스캔하지 않고 이 클래스를 참조하며, 정규화한 두 번째 패스는 트리거가 보였을 때에만 실행합니다. 스캔은 바이트마다 256개 항목 테이블을 한 번 조회하며, 코드는 sibuna-policy의 WAF 모듈에 있습니다. 누산기 네 개가 OR 연쇄를 짧게 유지하고, 두 바이트 쌍(/*와 인접한 공백 두 개)은 단일 바이트가 아직 정규화를 요구하지 않았을 때에만 찾습니다.

```
pub(crate) fn scan(text: &[u8]) -> Classes {
    let (chunks, remainder) = text.as_chunks::<8>();
    // 누산기 네 개로 OR 연쇄를 짧게 유지해 읽기를 연달아 발행할 수 있게 합니다.
    let mut lanes = [0u16; 4];
    for chunk in chunks {
        lanes[0] |= TABLE[usize::from(chunk[0])] | TABLE[usize::from(chunk[4])];
        lanes[1] |= TABLE[usize::from(chunk[1])] | TABLE[usize::from(chunk[5])];
        lanes[2] |= TABLE[usize::from(chunk[2])] | TABLE[usize::from(chunk[6])];
        lanes[3] |= TABLE[usize::from(chunk[3])] | TABLE[usize::from(chunk[7])];
    }
    let mut bits = lanes[0] | lanes[1] | lanes[2] | lanes[3];
    for &byte in remainder {
        bits |= TABLE[usize::from(byte)];
    }
    // 바이트 쌍은 아직 어떤 단일 바이트도 정규화를 요구하지 않았을 때에만 의미가 있습니다.
    if bits & CANONICAL == 0
        && ((bits & SPACE != 0 && has_pair(text, b' ', b' '))
            || (bits & (SLASH | STAR) == SLASH | STAR && has_pair(text, b'/', b'*')))
    {
        bits |= CANONICAL;
    }
    Classes { bits }
}
```

탐지기

- 시그니처. 모든 범주의 강한 시그니처는 태그가 붙은 오토마톤 하나에 있습니다. 적중하면 waf:sqli, waf:xss, waf:path-traversal, waf:rce라는 이름의 최종 deny가 됩니다.
- *SQL* 인젝션. 한 번 순회하는 단어 스캐너가 영숫자 단어(최대 7바이트)를 키워드 테이블에서 찾고, or / and에서는 literal = literal 항진식이 있는지 확인합니다. id=1 or 1=1은 최종 판정입니다. 그 밖의 경우에는 따옴표 바이트가 반드시 있어야 하며, 증거 점수가 다음 중에서 4점 이상이어야 합니다. 따옴표(1), 최대 두 개의 키워드(2), 주석 표식(1), =(1)입니다. admin'--는 그 자체로 최종 판정입니다. "it's a group order from the shop"은 3점이므로 통과합니다.
- 크로스 사이트 스크립팅. <가 있어야 합니다. 스크립트를 담을 수 있는 태그가 속성 위치의 on<event>= 속성과 함께 나타날 때 탐지합니다. onboarding=1은 해당하지 않습니다.
- 명령 인젝션. 구분자가 있어야 합니다. ;, |, &&, $(, 백틱, 줄 바꿈 뒤에 알려진 명령 이름이 오면 탐지합니다. status ok; done | next는 통과합니다.
- 경로 탐색. 시그니처(../, ..%2f, /etc/passwd, id_rsa, ...)와 NUL, %00입니다.

문법상 따옴표와 별표를 정당하게 포함하는 구조적 헤더(Accept*, Content-Type, Sec-*, If-*, ...)는 건너뜁니다. 경로, 쿼리, User-Agent, 그 밖의 헤더, 본문의 처음 8 KB를 검사합니다. 퍼센트 이

스케이프, 더하기 기호, 주석 시작 기호, 줄일 수 있는 공백이 있는 필드는 8 KB 스택 버퍼에 정규화한 뒤 다시 검사합니다. 그래서 `1%27/**/UnIoN/**/SeLeCt`와 `%252e%252e/`는 원래의 시그니처로 접힙니다.

비용

검사 비용은 요청 수만이 아니라 엔진에 들어오는 필드와 바이트에 달려 있습니다. 벤치마크 장은 Gate, Shield, 8 KB 본문의 경우를 따로 보고합니다. 8 KB 상한은 검사 한도일 뿐, 업로드의 나머지 부분이 안전하다는 주장이 아닙니다. forward-auth를 쓰는 인그레스는 관련 요청 정보를 전달해야 합니다. 전달되지 않은 바이트는 이 프로세스가 검사할 수 없습니다.

선언형 정책 엔진

> **학습 목표** 규칙 모델, 평가 순서, 임계값이 있는 WEIGH 점수, JSON 정책 문법을 읽어 봅니다.

규칙은 선택적인 조건(경로 패턴, User-Agent 패턴, 헤더 최대 4개, IPv4/IPv6 CIDR 블록 최대 8개, JA4 지문 또는 접두사 하나)의 논리곱이며, 동작 `ALLOW`, `DENY`, `CHALLENGE`, `WEIGH` 중 하나와 선택적인 규칙별 난이도(작업 비트)와 알고리즘, 부호 있는 `weight`를 가집니다. 평가 순서는 다음과 같습니다.

1. 의미 기반 WAF(Shield 전용): 위반은 최종 `deny`입니다.
2. 평판 트라이: `deny` 또는 `allow` 접두사는 최종 판정입니다.
3. 규칙(순서대로): 처음 일치한 최종 규칙의 결과를 반환하고, `WEIGH` 일치는 누적합니다.
4. 점수: 합계가 음수이면 `allow`, ≥ 40이면 `deny`, ≥ 10이면 `challenge`이며, 임계값을 넘는 5점마다 작업 비트를 하나씩 더하되 최대 여섯 비트까지 더합니다.
5. 정적 우회 경로, 그다음 트라이의 `challenge` 판정, 그다음 봇 오토마톤.
6. 기본 동작은 정책 파일이 달리 정하지 않으면 `challenge`입니다.

내장 규칙은 `/.well-known/*`, `/favicon.ico`, `/robots.txt`, `/__sibuna/*`를 허용하고, `CF-Worker` 클라이언트와 `Amazonbot`을 거부하며, 모든 `Mozilla` User-Agent에 챌린지를 제시합니다. 브라우저 문자열은 비용 없이 위조할 수 있으므로, 이를 허용하면 게이트는 장식에 불과해집니다.

```
{
  "default_action": "CHALLENGE",
  "waf": true,
  "thresholds": { "challenge_at": 10, "deny_at": 40, "bits_step": 5 },
  "ip_rules": { "10.0.0.0/8": "ALLOW", "2001:db8::/32": "DENY" },
  "rules": [
    { "name": "internal-api", "path": "/api/*", "headers": { "X-Api-Key": ".*" },
    "action": "ALLOW" },
    { "name": "checkout", "path": "/checkout/*", "action": "CHALLENGE",
      "challenge": { "difficulty": 20, "algorithm": "posw" } },
    { "name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight": 30 },
    { "name": "no-language", "headers": { "Accept-Language": "" },
    "action": "WEIGH", "weight": -5 }
  ]
}
```

따라가 보기: 유효한 세션을 지닌 인코딩된 공격

유효한 세션을 가진 클라이언트가 보낸 `GET /search?q=%2527%25200R%25201%253D1`을 생각해 봅시다. 먼저 HTTP 파서가 바이트를 복사하지 않고 경로와 쿼리를 분리합니다. 로컬 제한이 실행된 다음 Shield가 요청을 검사합니다. 퍼센트 기호 때문에 필드는 정규화 대상으로 표시됩니다. 한 번 디코딩하면 `%27%200R%201%3D1`이 드러나고, 두 번째 디코딩에서 `' OR 1=1`이 드러납니다.

이제 토크나이저는 따옴표, 키워드 `OR`, 서로 같은 리터럴 피연산자를 봅니다. 거부는 최종 정책 결과입니다. 유효한 세션이 있어도 이 결과는 허용으로 바뀌지 않습니다. 세션 확인은 챌린지 결정만 해소합니다. 저장소 훅은 크기가 제한된 인시던트 레코드를 복사하고, SQL은 나중에 실행됩니다. 응답 경로는 원본 서버 연결이 유휴 상태인 동안 엔진 스냅숏을 붙잡고 있을 필요가 없습니다.

연습 문제 6.1. 페이로드를 `O'Reilly`로, 그다음 `order=1`로 바꿔 보세요. 어느 쪽도 단독으로 SQL 인젝션을 입증해서는 안 되는 이유는 무엇일까요? 부분 문자열만 보는 탐지기가 낼 수 있는 오탐의 예를 드세요.

연습 문제 6.2. LDAP 인젝션 탐지기(예: `)(|(` 같은 패턴)를 추가하는 데 필요한 바이트 클래스 테이블 항목을 쓰고, 기존 클래스 비트 중 무엇을 재사용할 수 있는지 설명하세요.

불변식 설명하기. `Accept: */*`가 오탐이었던 이유, 새 SQL 탐지기가 요구하는 구조적 속성, 그리고 오토마톤과 토크나이저가 작업을 어떻게 나누는지 검토자에게 설명하세요.

제VII부

브라우저 작업 증명 엔진

서버 자신의 검증자 *crate*를 컴파일한 *WebAssembly* 모듈, *Web Worker* 프로토콜, *JavaScript* 대체 증명자, 그리고 중간 페이지를 살펴봅니다.

소스 하나, 타깃 두 개

학습 목표 이 장을 마치면 브라우저 모듈이 서버 자신의 `sibuna-pow` crate로부터 어떻게 빌드되는지 설명하고, 내보낸 함수의 이름을 대며, 순차 작업 증명자가 탭 안에서 필요로 하는 메모리를 설명할 수 있어야 합니다.

빌드 연결

브라우저 솔버는 서버가 검증에 쓰는 바로 그 소스를 컴파일합니다. `cmd/sibuna-pow-wasm`은 `wasm32-unknown-unknown`용으로 빌드하는 `cdylib` 유형의 `no_std` 라이브러리 crate이며, 의존성은 워크스페이스 crate인 `sibuna-pow`와 `sibuna-crypto`뿐입니다.

```
# cmd/sibuna-pow-wasm/Cargo.toml
[lib]
crate-type = ["cdylib", "rlib"]

[dependencies]
sibuna-crypto = { workspace = true }
sibuna-pow = { workspace = true }

# Cargo.toml(워크스페이스 루트): 모듈을 크기 우선으로 빌드합니다
[profile.release.package.sibuna-pow-wasm]
opt-level = "z"
debug = false
strip = true
```

데몬은 이 모듈을 내장합니다. 데몬의 빌드 스크립트(`cmd/sibuna/build.rs`)는 워크스페이스의 target 디렉터리(`target/wasm32-unknown-unknown/wasm/`)로 중첩된 `cargo rustc -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm`을 실행합니다. 중첩 빌드는 `wasm` 프로파일의 디렉터리만 잠그므로 바깥 빌드를 기다리지 않습니다. 최종 링크에는 `--compress-relocations`가 붙습니다. 스크립트는 결과가 WebAssembly 매직 바이트로 시작하는지 확인하고 빌드 출력 디렉터리로 복사합니다. WebAssembly 타깃이 없는 빌드에서는 `SIBUNA_POW_WASM=<path>`로 미리 빌드한 모듈을 대신 내장합니다. 그런 다음 데몬은 그 바이트를 바이너리에 포함합니다.

```
/// `build.rs`가 `cmd/sibuna-pow-wasm`에서 빌드한 브라우저 솔버.
pub const WASM_SOLVER: &[u8] =
    include_bytes!(concat!(env!("OUT_DIR"), "/sibuna-pow.wasm"));
```

```
/// 솔버의 Web Worker.
pub const WORKER_JS: &str = include_str!("../../../web/worker.js");
```

어긋날 수 있는 두 번째 구현이 없습니다. 모듈은 데몬이 `/__sibuna/wasm/sibuna-pow.wasm`에서 1시간 캐시 헤더(`Cache-Control: public, max-age=3600`)와 함께 제공합니다. 위 프로파일로 빌드한 모듈은 두 계층을 모두 포함해 14,744바이트입니다(`benchmarks/results/primitives-latest.json`의 `wasm_bytes`).

내보낸 함수

모듈은 함수 여섯 개를 내보내며, 워커 스크립트는 이름으로 이 함수들을 호출합니다.

내보낸 함수	용도
`sibuna_get_buffer_ptr()` / `sibuna_get_buffer_len()`	워커가 챌린지 문자열을 써 넣는 256바이트 입력 버퍼
`sibuna_solve_step(ptr, len, bits, start, max_steps)`	Hashcash: `start`부터 nonce `max_steps`개를 탐색하고, nonce 또는 모든 비트가 1인 표식 값을 반환합니다
`sibuna_solve_sha256(ptr, len, bits)`	nonce 0부터 한 번의 배치로 실행하는 Hashcash(`u32::MAX` 단계로 호출한 `sibuna_solve_step`)
`sibuna_posw_solve(len, depth, challenges)`	순차 작업: 버퍼에 대해 증명자를 실행하고 증명 길이를 반환하며, 인수가 잘못되면 0을 반환합니다
`sibuna_posw_proof_ptr()`	증명자 작업 공간 안에서 증명 바이트가 시작하는 위치

증명자의 작업 공간, 즉 보관하는 상위 레벨, 형제 스택, 증명은 약 92 KiB의 정적 영역입니다. 할당자는 링크되지 않으며 모듈에는 가져오기(import)가 없습니다. `Workspace::new`는 `const fn`이므로 이 정적 영역은 모듈 안의 초기화된 데이터가 아니라 0으로 채운 메모리만 차지합니다.

구현: `sibuna_solve_step` Hashcash 내보내기 함수입니다. 선형 메모리에서 챌린지를 빌려 공유 nonce 탐색을 한 배치 실행합니다. `cmd/sibuna-pow-wasm/src/lib.rs`(crate `sibuna-pow-wasm`)에 있습니다.

```
#[unsafe(no_mangle)]
pub unsafe extern "C" fn sibuna_solve_step(
    challenge_ptr: *const u8,
    challenge_len: usize,
    difficulty_bits: u32,
    start_nonce: u64,
    max_steps: u32,
) -> u64 {
    let challenge: &[u8] = if challenge_len == 0 {
        &[]
    } else {
        // SAFETY: 호출자는 `challenge_ptr`에 읽을 수 있는 `challenge_len`바이트가 있고
        // 그 바이트가 호출이 끝날 때까지 바뀌지 않음을 보장합니다.
        unsafe { core::slice::from_raw_parts(challenge_ptr, challenge_len) }
```

```
    };
    Search::new(challenge)
        .find(difficulty_bits, start_nonce, u64::from(max_steps))
        .unwrap_or(EXHAUSTED)
}
```

내보낸 함수들은 이 crate의 외부 함수 경계이며, `unsafe` 코드가 있는 유일한 곳입니다. 입력 버퍼와 증명자 작업 공간은 `ExportCell` 안의 정적 변수입니다. 이 래퍼는 `UnsafeCell`을 감싸며, 이유를 밝힌 채 `Sync`로 선언되어 있습니다. 공유 메모리가 없는 WebAssembly 인스턴스는 스레드 하나로 실행되며, 각 내보내기 함수는 참조를 만들고 사용한 뒤 반환하기 전에 버리므로, 셀을 가리키는 참조는 언제나 많아야 하나만 살아 있습니다. 테스트가 내보내기 함수를 직접 호출하는 호스트에서는 테스트 전용 뮤텍스가 호출을 직렬화합니다. 패닉은 메시지를 포맷하지 않고 트랩(`unreachable` 명령어)으로 끝나므로, 모듈에는 포맷 코드와 가져오기가 없습니다. `sibuna-pow`의 함수는 모든 입력에 대해 정의된 전함수(total function)이고 결코 패닉하지 않으므로, 구조상 패닉에는 도달할 수 없습니다.

접두사 사전 해싱

Hashcash 안쪽 루프는 챌린지와 콜론을 한 번만 흡수하고 nonce마다 SHA-256 상태를 복제하므로, 후보 하나의 비용은 압축 두 번이 아니라 한 번입니다. 탐색은 브라우저 모듈과 네이티브 참조 솔버가 공유하는 `sibuna_pow::hashcash::Search`입니다.

```
pub fn new(challenge: &[u8]) -> Self {
    let mut prefix = Sha256::new();
    prefix.update(challenge);
    prefix.update(b":");
    Self { prefix }
}

pub fn digest(&self, nonce: u64) -> Digest {
    let mut buf = [0u8; NONCE_DIGITS_MAX];
    let mut hasher = self.prefix.clone(); // 흡수한 접두사를 값으로 복사
    hasher.update(format_decimal(nonce, &mut buf));
    hasher.finalize()
}

pub fn find(&self, bits: u32, start: u64, steps: u64) -> Option<u64> {
    let mut nonce = start;
    for _ in 0..steps {
        if meets_bits(&self.digest(nonce), bits) {
            return Some(nonce);
        }
        nonce = nonce.wrapping_add(1);
    }
    None
}
```

해셔 상태는 약 110바이트의 평범한 값이므로 복제 비용이 작습니다. 순차 작업 증명자도 같은 기법을 써서 명제 χ를 `base_hasher`에 한 번 흡수하고 노드마다 복제합니다. 이제 검증자도 그렇게 합니다. 명제를 한 번 흡수한 상태에서 챌린지로 지정된 잎을 각각 도출하므로, 같은 다이제스트를 얻으면서 검증에서 압축 t번을 아낍니다.

서버에서는 압축 함수가 실행 시점에 감지한 x86-64 SHA 확장이나 ARMv8 SHA-2 명령어를 사용합니다. 브라우저에는 그런 명령어가 없으므로 솔버는 거의 모든 시간을 이식 가능한 압축 함수(crates/crypto/src/sha256/portable.rs)에서 보냅니다. 이 함수는 WebAssembly를 위해 작성되었습니다. 64라운드를 펼쳐 놓았고, 메시지 스케줄은 16워드 창에 두어 16라운드부터는 각 라운드가 더는 필요 없는 워드를 덮어쓰며, 작업 변수 여덟 개는 옮기는 대신 라운드마다 이름을 바꿉니다.

```
fn compress_block(state: &mut State, block: &Block) {
    let mut w = [0u32; 16];
    for (word, bytes) in w.iter_mut().zip(block.as_chunks::<4>().0) {
        *word = u32::from_be_bytes(*bytes);
    }
    let [mut a, mut b, mut c, mut d, mut e, mut f, mut g, mut h] = *state;
    eight_rounds!(w, 0, a, b, c, d, e, f, g, h);
    eight_rounds!(w, 8, a, b, c, d, e, f, g, h);
    // ... 라운드 16부터 55까지
    eight_rounds!(w, 56, a, b, c, d, e, f, g, h);
    for (word, value) in state.iter_mut().zip([a, b, c, d, e, f, g, h]) {
        *word = word.wrapping_add(value);
    }
}

/// 스케줄 워드 W(i)를 반환합니다. 라운드 16부터는 제자리에서 계산합니다.
#[inline(always)]
fn schedule(w: &mut [u32; 16], i: usize) -> u32 {
    if i >= 16 {
        let w15 = w[(i + 1) % 16];
        let w2 = w[(i + 14) % 16];
        let s0 = w15.rotate_right(7) ^ w15.rotate_right(18) ^ (w15 >> 3);
        let s1 = w2.rotate_right(17) ^ w2.rotate_right(19) ^ (w2 >> 10);
        w[i % 16] = w[i % 16]
            .wrapping_add(s0)
            .wrapping_add(w[(i + 9) % 16])
            .wrapping_add(s1);
    }
    w[i % 16]
}
```

eight_rounds! 하나는 변수 이름을 한 바퀴 완전히 돌립니다. 매크로는 라운드마다 이름을 한 자리씩 회전시켜 넘기므로, 한 라운드는 두 변수에만 쓰고 나머지 여섯 개는 옮기지 않습니다. 라운드 인덱스가 리터럴이므로 모든 스케줄 위치가 상수가 되고, 블록 전체가 값이 WebAssembly 지역 변수에 머무는 직선 코드가 됩니다. 이 배치는 펼치지 않은 루프로 64워드 스케줄을 계산하는 방식보다 V8에서 약 20%, 네이티브에서 약 14% 빠릅니다(Rust 구현으로 측정, 2026년 10월).

워커 프로토콜

> **학습 목표** 메시지 프로토콜을 읽고, 대체 경로의 순서를 이해하며, WebAssembly와 JavaScript 사이의 부호 있는 정수 경계가 일으킨 표식 값 버그를 살펴봅니다.

페이지는 web/worker.js에 메시지 하나를 보내고 진행, 대체, 결과 메시지를 받습니다.

```
// 페이지 -> 워커
{ challenge: spec.id, algorithm: spec.algorithm, difficulty: spec.difficulty,
    challenges: spec.challenges }

// 워커 -> 페이지
{ type: 'progress', iterations: 40000 }
{ type: 'fallback', message: 'Wasm unavailable; running JS prover' }
{ type: 'solved', challenge: spec.id, solution: { nonce: '90766' } }  // hashcash
{ type: 'solved', challenge: spec.id, solution: { proof: '<base64url>' } } // posw
{ type: 'error', message: '...' }
```

워커는 먼저 WebAssembly를 시도하고, `WebAssembly`가 없거나 정책으로 막혀 있거나 인스턴스 생성에 실패하면 순수 JavaScript로 대체합니다. JavaScript 증명자는 `sibuna-pow`의 Rust 증명자를 라벨 하나하나까지 그대로 따르며 바이트 단위로 동일한 출력을 냅니다. 테스트는 두 계층 모두에 대해 고정된 nonce와 증명을 확인하며, SID 0010은 솔버가 헤드리스 브라우저에서 WebAssembly로도, JavaScript 대체 경로로도 통과하도록 요구합니다. JavaScript 증명자는 훨씬 느리지만 모든 브라우저가 통과할 수 있음을 보장합니다.

부호 있는 표식 값 WebAssembly는 64비트 정수를 JavaScript에 부호 있는 `BigInt`로 반환합니다. Hashcash 내보내기 함수는 "배치 소진"을 모든 비트가 1인 값(`EXHAUSTED`, `u64::MAX`)으로 알리는데, JavaScript는 이를 `18446744073709551615n`이 아니라 `-1n`으로 봅니다. 원래 워커는 부호 없는 값과 비교했으므로, 루프가 첫 배치 뒤에 엉터리 nonce를 가지고 끝났습니다. 수정은 `BigInt.asUintN(64, result)` 호출 한 번이며, 워커에는 지금도 이 호출이 남아 있습니다.

```
// WebAssembly i64 결과는 부호 있는 BigInt로 도착하므로, 모든 비트가 1인
// u64 "소진" 표식 값은 -1n으로 읽힙니다.
const found = BigInt.asUintN(64,
    ex.sibuna_solve_step(ptr, encoded.length, bits, nonce, maxSteps));
if (found !== 18446744073709551615n) return found.toString();
```

이 교훈은 일반적으로 적용됩니다. 경계를 넘는 모든 u64는 정규화해야 합니다. Rust는 JavaScript 쪽에서 이를 강제할 수 없습니다. 내보내기 함수의 타입은 `u64`이고, 변환은 엔진 안에서 일어납니다.

보정

학습 목표 난이도 설정을 브라우저 엔진과 네이티브 실리콘에서의 실제 경과 시간과 연결합니다.

노트북 한 대의 평균 풀이 시간으로 난이도를 고르지 마세요. Hashcash에는 기하 분포의 꼬리가 있고, 브라우저의 시간에는 스케줄링, 발열 상태, 시작과 컴파일 시간이 포함됩니다. 배포를 보정할 때는 브라우저 버전, 기기, 알고리즘, 매개변수, 그리고 반복 풀이의 분포를 기록해야 합니다. 모듈을 가져오고 컴파일하는 시간은 정상 상태의 해싱과 분리하세요.

풀이 예제: 보정 계획 b를 고정하고, 모듈을 예열한 뒤, 대상 기기 등급마다 독립적으로 발급된 퍼즐을 100개 이상 풉니다. 중앙값과 높은 백분위수의 소요 시간, 실패, 증명 크기를 기록합니다. $b+1$에서 반복합니다. 기대 시행 횟수는 두 배가 되지만 개별 표본은 그렇지 않을 수 있습니다. 느린 기기의 예산에 맞춰 설정을 고른 다음, 가져오기, 풀이, 검증의 전체 흐름을 시험합니다.

저장소의 네이티브 기본 연산 측정은 휴대폰이나 브라우저의 풀이 시간을 약속하는 근거가 되지 않습니다. WASM 모듈은 같은 산술을 실행하면서도 시작, 실행, 메모리 비용이 다를 수 있습니다. 네이티브 서버는 SHA-NI로 해시하지만, 브라우저는 위의 이식 가능한 압축 함수를 실행합니다. 워커의 메시지는 서버의 요청 처리기에 타이머를 추가하지 않고도 이 비용을 측정할 수 있는 경계입니다.

중간 페이지

페이지(`web/challenge.html`, 데몬에 내장)는 카드, 진행률을 표시하지 않는 진행 막대, 상태 줄로 이루어진 HTML 파일 하나입니다. 자신의 위치에 대한 챌린지를 요청하고(`?path=`), 워커를 띄우고, `fetch`로 풀이를 보낸 뒤, 성공하면 다시 불러옵니다. `fetch` 호출은 지터를 넣은 지수 백오프로 재시도하며, `4xx` 검증 응답은 진단 제목과 함께 표시됩니다. 페이지의 `Content-Security-Policy`는 정확히 스크립트 하나, 즉 솔버 블록만 SHA-256 해시로 허용하며, 데몬은 내장된 페이지에서 이 해시를 한 번 계산합니다. 화면 밖에 둔 허니팟 링크(`/__sibuna/honeypot`)는 사람에게도, 스크립트를 실행하는 브라우저에게도 보이지 않습니다. 모든 `href`를 따라가는 맹목적인 크롤러는 스스로 차단됩니다.

연습 문제 7.1. WebAssembly 증명자가 노트북에서 기본 난이도의 챌린지를 20 ms에 풀고, JavaScript 순차 증명자는 50배 느리다고 합시다. 그 노트북과 네 배 느린 휴대폰에서 대체 경로의 풀이 시간을 추정하고, 대체 경로를 보고하는 클라이언트를 위한 정책을 제안하세요(힌트: 워커는 페이지가 전달할 수 있는 `fallback` 메시지를 보냅니다).

불변식 설명하기. 서버의 검증자 소스로 브라우저 모듈을 컴파일하는 것이 두 구현을 서로 대조해 테스트하는 것보다 강한 보장인 이유를 설명하세요.

제VIII부

실측 평가

이 부에서는 저장소에 커밋된 하니스로 기록한 *Sibuna*의 측정 결과를 제시합니다. 기본 연산 지연, 외부 부하 생성기로 부하를 걸고 *CPU*와 메모리 사용량을 함께 기록한 전체 제품 측정, 개방 루프 생성기로 같은 부하를 걸었을 때의 지연, 그리고 널리 배포된 작업 증명 게이트인 *Anubis*와의 비교입니다. 실시간 대시보드 *8*개를 띄운 콘솔 실행이 기록을 마무리합니다. 이 부의 모든 수치는 커밋된 결과 파일이나 설계 기록에서 가져왔으며, 각 표는 해당 결과 파일과 그 생성 시각을 밝힙니다. 기능 검사 통과와 성능 측정은 서로 다른 질문에 답합니다. 한 호스트에서 얻은 측정 결과가 다른 호스트에서도 같은 순위를 뜻하지는 않습니다.

방법론

학습 목표 이 장을 마치면 이 부의 모든 수치를 재현하고, 각 하니스가 무엇을 포함하고 무엇을 제외하는지 말하고, 변동 범위 열의 의미를 설명하고, 직접 측정한 수치와 기록된 수치나 발표된 수치를 구별할 수 있어야 합니다.

하니스는 Rust 도구 하나인 `tools/bench` (바이너리 `sibuna-bench`)의 명령이며, `cargo xtask bench`로 실행합니다. 콘솔 실행은 별도 도구가 기록합니다.

표 11 하니스와 결과 파일(모두 `benchmarks/results/` 아래)

하니스	측정 대상	결과 파일
`cargo xtask bench primitives`	CPU 하나에서 실행하는 기본 연산 워크로드 16개와 해시 기반 속도 제한기 경로의 행 2개: 배치 7개, 연산당 중앙값과 최소-최대 범위	`primitives-latest.json`, `primitives-latest.md`
`cargo xtask bench product`	`wrk`로 부하를 건 전체 데몬: forward-auth 모드와 리버스 프록시 모드에서 Gate와 Shield의 처리량, 지연 백분위, 요청당 CPU 시간, 최대 상주 메모리	`product-latest.json`, `product-latest.md`
`cargo xtask bench latency`	개방 루프 생성기로 고정된 제공 요청률을 걸었을 때 허용된 요청의 지연 백분위	`latency-latest.json`, `latency-latest.md`
`cargo xtask bench product --anubis`	Anubis를 추가 제품으로 넣고 `caddy respond`를 원본 서버로 둔 전체 제품 워크로드	`comparison-latest.json`, `comparison-latest.md`
`cargo xtask bench admission --anubis`	keep-alive 클라이언트 2개로 측정한, forward-auth 모드에서 Sibuna와 Anubis의 초당 요청 허용 연산 수	`admission-latest.json`, `admission-latest.md`

하니스	측정 대상	결과 파일
sibuna-browser dashboards (tools/browser)	기능 검사용 콘솔 실행: 생성기가 트래픽을 보내는 동안 헤드리스 Chromium에서 실시간 대시보드 8개를 실행	console-8-dashboards-2026-10-15.md

.md 파일은 요약입니다. .json 파일은 모든 반복, 머신, 리비전, 그리고 측정한 모든 바이너리의 경로, 크기, SHA-256을 담습니다. 이력은 Git이 보관하므로 디렉터리에는 최신 파일만 둡니다. 일부 파일은 하니스의 이전 버전으로 기록되었으며, 같은 실행에서 측정한 두 번째 데몬 빌드의 행도 담고 있습니다. 이 부에서는 Sibuna의 행을 보고합니다.

기본 연산

각 워크로드는 시간을 재지 않는 예열(최대 1,000회 연산) 뒤에 시간을 재는 배치 7개로 실행합니다. 변경 가능한 상태(사용 완료 집합, 속도 제한기)는 각 배치 전에 타이머 밖에서 다시 만듭니다. 타이머는 개별 연산이 아니라 배치 전체를 감쌉니다. 연산마다 시계를 읽으면 그 비용이 측정 대상인 100 ns 미만의 작업만큼 들기 때문입니다. 따라서 연산별 백분위는 보고하지 않으며, 변동 범위는 배치 사이의 범위입니다. 연산당 값으로는 중앙값 배치를 보고하고, 가장 빠른 배치와 가장 느린 배치를 옆에 적습니다.

측정 모음은 CPU 하나에 고정된 자식 프로세스(sibuna-bench primitives-run)에서 실행됩니다. 공유 호스트에서는 바쁜 SMT 형제나 낮아진 클럭이 측정 모음 실행 전체를 느리게 합니다. 그래서 도구는 측정 모음을 세 번 실행하고(--rounds), 행마다 중앙값이 가장 낮은 라운드를 남깁니다. 라운드 안의 통계는 위와 같습니다.

전체 제품

각 모드(forward auth, 리버스 프록시)와 각 운영 구성(Gate, Shield)마다 하니스는 데몬을 메모리 모드로 --algorithm hashcash --difficulty 8 --rate-limit 100000000 --idle-timeout 5와 워커 4개(CPU 집합의 CPU당 하나)로 시작합니다. 그다음 브라우저처럼 sibuna-pow로 데몬의 Hashcash 챌린지를 풀어 세션을 얻습니다. 이후 keep-alive 연결에서 wrk로 네 가지 워크로드를 실행합니다. 스레드 2개, 연결 64개, 5초, 반복 3회입니다.

- admitted: 유효한 세션 쿠키가 있는 GET /private입니다. 리버스 프록시 모드에서는 요청이 원본 서버에 도달하고 그 응답이 중계됩니다. forward-auth 모드에서는 데몬이 직접 200으로 답합니다.
- challenged: 같은 요청을 쿠키 없이 브라우저 User-Agent로 보냅니다. 데몬은 챌린지를 제공합니다. 리버스 프록시 모드에서는 HTML과 함께 200, forward-auth 모드에서는 401입니다.
- allowed_static: curl/8.0이 보내는 GET /robots.txt이며, 두 운영 구성 모두 챌린지 없이 허용합니다.
- attack: 유효한 세션에 q=' OR 1=1--를 더한 요청입니다. Shield만 이를 거부합니다(403). Gate는 이를 허용하고, 리버스 프록시 모드에서는 원본 서버로 중계합니다.

원본 서버는 스텁(sibuna-bench origin)입니다. 표준 라이브러리만 쓰고, 연결당 스레드 하나를 두며, keep-alive를 지원하고, 41바이트 페이지를 돌려줍니다. 요청당 CPU 시간은 실행 동안 /proc/<pid>/stat에서 읽은 데몬의 사용자 시간과 시스템 시간의 합(10 ms 틱)을 wrk가 센 요청 수로 나눈 값입니다. 원본 서버의 CPU는 포함하지 않습니다. 최대 메모리는 /proc/<pid>/status의 VmHWM입

니다. 각 실행이 자기 최댓값을 보고하도록 실행마다 /proc/<pid>/clear_refs로 초기화합니다. 모든 수치는 반복의 중앙값이며, 최솟값과 최댓값은 JSON 파일에 함께 보관합니다.
wrk는 폐쇄 루프 생성기로, 요청 64개를 항상 처리 중인 상태로 유지합니다. 따라서 더 빠른 데몬이 더 많은 요청을 받습니다. 또한 wrk의 백분위에는 느린 응답을 기다리는 동안 보내지 못한 요청이 빠집니다(조정된 누락, coordinated omission). 그래서 그 p99는 주어진 부하에서의 지연이 아니라 데몬이 스스로 포화되었을 때의 동작을 보여 줍니다. 그 질문에는 지연 하니스가 답합니다.

같은 부하에서의 지연

개방 루프 생성기(sibuna-bench generate)는 연결마다 블로킹 스레드 하나를 실행합니다. 연결 64개는 각각 고정된 일정에 따라 세션과 함께 GET /private를 보냅니다. 한 연결의 요청 k는 이전 응답과 무관하게 $t_0 + o + kT$에 보내야 합니다. 여기서 o는 연결의 오프셋이고 T는 그 간격입니다. 다음 요청을 보낼 시각이 지난 뒤에 응답이 도착하면, 그 요청은 즉시 나가고 지연은 wrk2처럼 보냈어야 할 시각부터 셉니다. 예정 시각에 연결이 유휴 상태였다면 지연은 실제로 보낸 시각부터 셉니다. 그래서 생성기 자신의 sleep 초과 시간은 데몬의 몫으로 계산되지 않습니다.

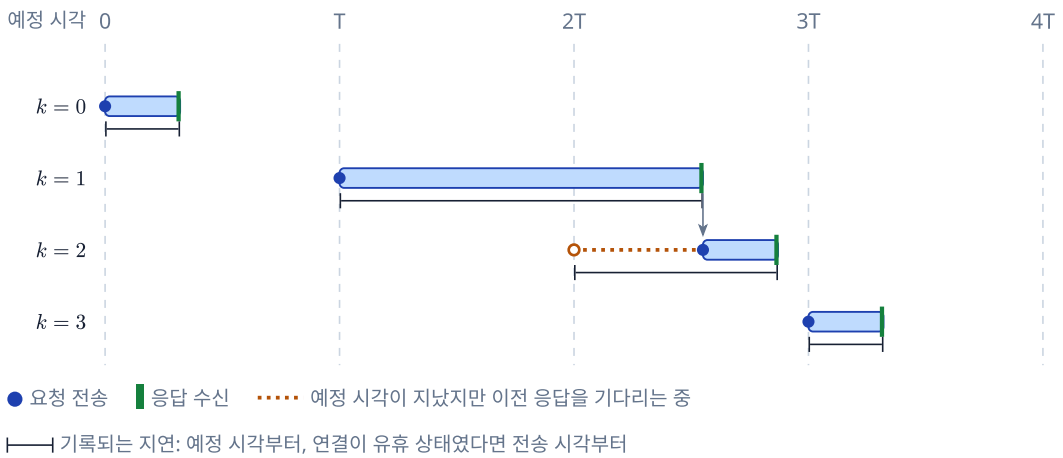


그림 10 개방 루프 생성기의 연결 하나입니다. 요청 1의 응답은 요청 2의 예정 시각 뒤에 오므로 요청 2는 즉시 전송되고 그 지연은 2T부터 셉니다. 요청 3은 연결이 유휴 상태일 때 보내므로 전송 시각부터 셉니다.

하니스는 Shield 운영 구성과 허용 워크로드를 제공 요청률 다섯 가지, 즉 초당 10,000, 25,000, 50,000, 100,000, 150,000 요청으로 실행합니다. 각 요청률은 2초 예열, 10초 측정, 실행 3회입니다. 하니스는 한 번의 실행에서 데몬 빌드 두 개를 측정할 수 있습니다. 이때 두 빌드를 같은 CPU 집합에서 시작하고, 모든 요청률과 반복마다 번갈아 측정합니다. 쉬는 쪽은 CPU를 쓰지 않으므로 머신의 느린 변화가 두 빌드에 똑같이 영향을 줍니다. 기록된 실행도 이렇게 했습니다. 백분위는 한 실행에서 측정한 모든 요청에 대한 최근접 순위(nearest-rank) 값입니다. 각 수치는 세 실행의 중앙값이며, p99의 최소-최대를 옆에 적습니다.

CPU 집합

CPU 집합은 다른 어떤 설정보다 중요합니다. 데몬은 CPU 0부터 3에서 실행합니다(--cpus; --workers는 이 집합의 크기입니다). wrk, 지연 생성기, 원본 서버는 CPU 8부터 15에서 실행합니다(--load-cpus). 기본 연산 측정 모음은 CPU 4에서 실행합니다(--cpu). 모든 프로세스는 시작 전에

taskset -c로 고정하므로, 프로세스가 만드는 모든 스레드가 그 집합을 물려받습니다. 요청 허용 하니스는 제품에 CPU 0과 1 두 개를, 클라이언트에 CPU 8부터 15를 씁니다.



그림 11 AMD Ryzen 9 5950X의 CPU 집합입니다. CPU 0부터 7은 첫 번째 CCD의 코어 8개, 8부터 15는 두 번째 CCD의 코어 8개, 16부터 31은 그 SMT 형제입니다. 데몬과 부하 생성기는 코어를 공유하지 않습니다.

다른 머신에서는 코어를 공유하지 않는 집합을 고릅니다(lscpu -e가 토폴로지를 보여 줍니다).

머신과 조건

파일은 paxos-zig에서 기록했습니다. AMD Ryzen 9 5950X 16-Core Processor(논리 CPU 32개), Linux 7.0.0-28-generic x86_64, 부스트를 켠 powersave 거버너, rustc 1.99.0입니다. 기본 연산, 전체 제품, 지연 파일은 2026년 10월 15일에, Anubis와의 비교는 2026년 10월 16일에 기록했습니다. 데몬은 cargo build --release -p sibuna, 즉 SQLite 저장소와 콘솔을 포함해 컴파일한 기본 빌드입니다(SID 0010). 전체 제품 파일과 지연 파일에 적힌 바이너리는 49,113,856바이트입니다. 각 파일은 리비전(기본 연산은 efce14d, 전체 제품과 지연 실행은 249d913)과 dirty: true를 기록합니다. 후자의 작업 트리에는 SID 0002가 기록한 연결 스레드 재사용이 이미 들어 있었고, SID 0002는 그 효과의 근거로 latency-latest.md를 인용합니다. 비교 파일과 요청 허용 파일은 리비전 54e4ec5와 dirty: true를 적습니다. 작업 트리에 tools/bench의 하니스 추가분이 있었기 때문입니다. 이 파일들의 데몬은 그 리비전의 릴리스 빌드(37,450,232바이트)입니다. 1분 부하 평균은 비교 실행 전에 3.53, 요청 허용 실행이 끝났을 때 0.56이었습니다.

파일에 기록된 1분 부하 평균은 기본 연산 실행 때 2.46, 지연 실행 전 1.30과 후 52.31, 전체 제품 실행 전 52.31과 후 33.39입니다. 높은 수치는 지연 생성기의 스레드 64개 때문이며, 전체 제품 실행은 지연 실행 직후에 이어졌습니다.

> **측정 범위** 로컬 측정만 보고합니다. 모든 실행은 호스트 하나에서 루프백으로 이루어집니다. TLS, WAN, 실제 브라우저는 없습니다. 데몬은 저장소 없이 메모리 모드로 실행합니다.

타이머는 배치를 감싸며, 상태 초기화와 예열은 시간을 재지 않습니다. 하니스는 할당 횟수를 세지 않습니다(파일에는 "not instrumented; API/source audit only"라고 적혀 있습니다). 대신 요청 경로에 할당이 없다는 점은 제V부의 정상 상태 테스트가 확인합니다. 어떤 하니스도 데몬에 링크되거나 요청 경로에 훅을 추가하지 않습니다. 전체 제품 하니스와 지연 하니스는 수정하지 않은 릴리스 바이너리를 실행하며, 이 바이너리는 SHA-256으로 식별합니다. 이 호스트에서 매긴 순위가 다른 호스트에서도 그대로 성립하지는 않습니다.

재현 방법

Linux, `taskset` (util-linux), 그리고 `PATH`에 있는 `wrk`가 필요합니다.

```
cargo xtask bench primitives
cargo xtask bench product
cargo xtask bench latency
cargo xtask bench help
```

`cargo xtask bench`는 먼저 `cargo build --release -p sibuna`로 데몬을 빌드한 다음 `sibuna-bench`를 릴리스 모드로 실행합니다. 도구는 자기 바이너리 옆에서 데몬을 찾으며, `--rust <path>`로 다른 경로를 지정할 수 있습니다. `cargo xtask bench help`는 옵션을 나열합니다. 그중에는 `--cpus`, `--load-cpus`, `--cpu`, `--rounds`, `--repetitions`, `--seconds`, `--rates`, `--warmup`이 있습니다. Anubis와의 비교를 기록한 명령은 아래 Anubis 절에 있습니다.

기본 연산 지연

표는 기본 연산 실행의 모든 행을 보여 줍니다. JSON 파일은 각 행을 하위 시스템과 워크로드 이름으로 부릅니다(예: `pow_verify`와 `hashcash_16_bits`).

표 12 Ryzen 9 5950X 코어 하나에서 측정한 기본 연산의 연산당 시간(나노초) (`benchmarks/results/primitives-latest.md`, 생성 2026-10-15T22:34:59Z)

워크로드	ns/연산	최소-최대
Hashcash 검증(16비트)	70.2	69.9-71.0
PoSW 검증(깊이 13, t = 16)	22272.4	22257.0-22316.4
봇 시그니처(40개, Aho-Corasick)	55.7	55.3-57.2
봇 시그니처(40개, 순차 검색)	763.0	762.2-763.5
IPv4 CIDR 조회	32.0	31.8-32.2
IPv6 CIDR 조회	79.3	78.5-79.6
세션 토큰(키 기반 BLAKE3)	155.8	155.3-156.4
세션 토큰(Ed25519)	40587.5	40544.6-40631.1
사용 완료 집합(Robin Hood), 사용 처리와 조회	21.0	20.8-21.3
속도 제한기(GCRA), 클라이언트 바이트	12.8	12.7-12.8

워크로드	ns/연산	최소-최대
속도 제한기, 규칙 범위 4개, 클라이언트 바이트	13.1	13.0-13.2
속도 제한기(GCRA), 해시된 클라이언트	6.2	6.1-6.3
속도 제한기, 규칙 범위 4개, 해시된 클라이언트	6.3	6.2-6.4
HTTP 파싱과 쿠키	132.5	132.3-133.2
분류, Shield 프로파일	790.8	787.2-793.1
분류, Gate 프로파일	75.9	75.6-76.1
WAF 본문 스캔(8 KiB)	12784.3	12771.5-12809.2
WAF 본문 스캔(8 KiB), SQL 인젝션 감사 모드	15485.4	15460.9-15530.1

변동 범위는 좁습니다. 어느 행에서도 중앙값과 2.8% 넘게 차이 나는 배치는 없습니다. 그림은 같은 행을 로그 눈금으로 보여 주므로, 6 ns인 행과 40 us인 행을 나란히 놓고 읽을 수 있습니다.

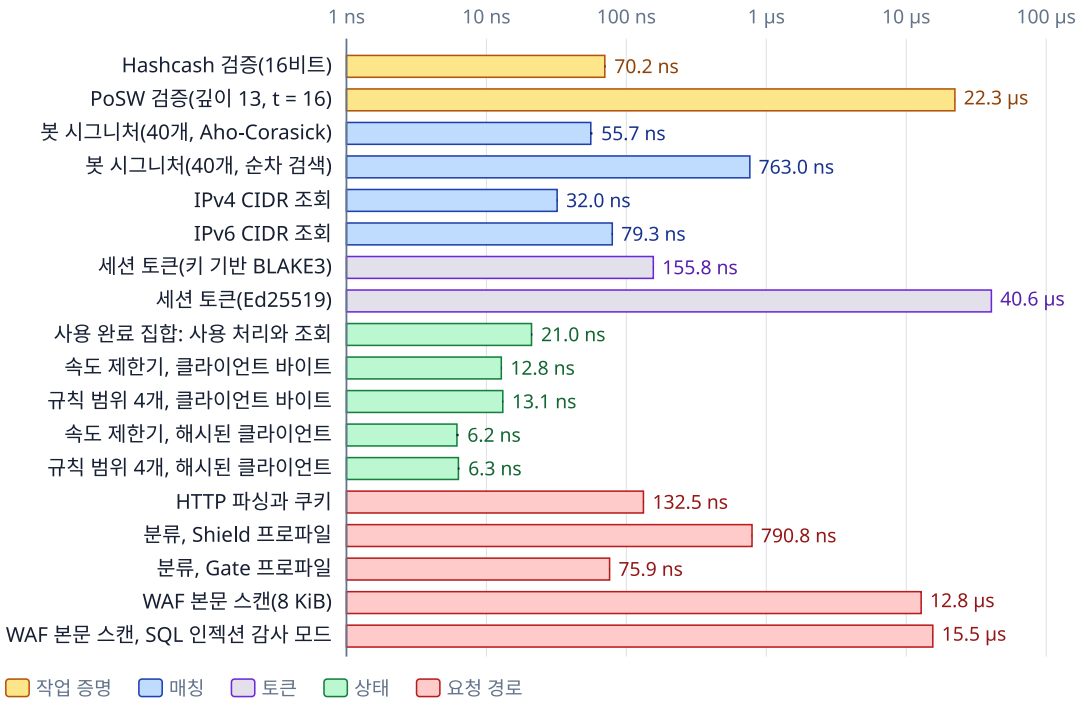


그림 12 benchmarks/results/primitives-latest.json에서 읽어 로그 눈금으로 그린 기본 연산 측정 시간.

작업 증명 검증

Hashcash 검증은 챌린지, 구분자, 십진수 nonce를 해시하며 70.2 ns가 걸립니다. 순차 작업 검증기는 그래프 개방을 검사하며 22,272.4 ns가 걸립니다. 입력 길이가 다르므로 해시 호출 수와

SHA-256 압축 횟수도 Hashcash와 다릅니다. 두 행 모두 SHA-256이 비용을 좌우합니다. `sibuna-crypto`는 CPU가 SHA-NI 명령어를 지원하면 이를 실행 시점에 감지해 사용합니다(SID 0010).

매칭

밀집 Aho-Corasick 오토마톤과 순차 부분 문자열 검색은 같은 패턴과 같은 any-match 의미(하나라도 일치하면 성공)를 사용합니다. 같은 시그니처 40개에서 오토마톤은 순차 검색보다 13.7배 빠릅니다(55.7 ns 대 763.0 ns). 오토마톤의 테이블은 상태마다 열 256개를 두는 대신 바이트 클래스를 사용합니다(SID 0006, Part F). 그 덕분에 봇 시그니처 테이블은 42,244바이트에 머뭅니다(아래 자원 사용량 표). CIDR 분류기는 32.0 ns(IPv4)와 79.3 ns(IPv6)가 걸립니다.

토큰

BLAKE3 MAC 토큰은 155.8 ns에 검증됩니다. `sibuna-crypto`는 SIMD 경로 없이 이식 가능한 방식으로 BLAKE3를 구현합니다. SIMD 경로는 큰 입력에서 유리하지만, Sibuna가 BLAKE3로 해시하는 입력은 100바이트 미만이며 여기서는 이식 가능한 코드도 빠릅니다. Ed25519 압축 토큰은 40,587.5 ns에 검증됩니다. 검증기는 너비 5와 너비 8의 NAF 자릿수를 쓰고, 디코딩한 공개 키를 보관합니다. Ed25519 검증 한 번의 비용은 MAC 검증 약 260번과 비슷합니다. MAC은 Sibuna의 기본 토큰 방식입니다.

상태

사용 완료 집합 행은 새 태그를 삽입하고 다시 조회합니다. 21.0 ns가 걸립니다. 이 행의 테이블은 슬롯 4,096개짜리 샤드 16개(태그 65,536개)로 이루어집니다. 데몬의 기본값은 1,048,576개입니다. 테이블은 각 배치 전에 타이머 밖에서 제자리에서 비웁니다. 따라서 모든 배치는 상주 메모리를 탐색합니다. 이전 버전의 측정 모음은 배치마다 데몬 기본 크기로 0으로 초기화한 새 테이블을 만들었습니다. 그 측정은 첫 쓰기 때 커널이 테이블 페이지를 매핑하는 시간을 쟀습니다. 이는 실행 중인 데몬이 한 번만 치르는 비용이며, 우리가 보고하는 행이 아닙니다.

속도 제한기 행은 주의해서 읽어야 합니다. 클라이언트 바이트에 대한 두 행(`gcra_check`와 `gcra_four_rule_scopes`)은 바이트 API를 호출합니다. 각 검사는 클라이언트 주소를 바이트로 받아, 먼저 운영 체제 난수에서 얻은 비밀 키로 키 기반 SipHash-1-3을 계산합니다. 비밀 키 덕분에 누구도 피해자와 같은 탐사 윈도를 공유하는 주소를 계산할 수 없습니다(SID 0010과 `sibuna-store` 문서). 이 두 행은 12.8 ns와 13.1 ns가 걸립니다.

데몬은 요청 경로에서 바이트 API를 쓰지 않습니다. 요청마다 클라이언트를 한 번 해시하고(`ClientHasher`), 그 해시를 참조하는 모든 테이블에 넘깁니다. 해시된 클라이언트에 대한 두 행(`gcra_check_hashed`와 `gcra_four_rule_scopes_hashed`)은 해시를 루프 전에 계산해 두고 그 경로를 측정합니다. 검사당 6.2 ns와 6.3 ns입니다. 바이트 행과 해시 행의 중앙값 차이인 6.6 ns는 이 입력에서 키 기반 해시의 비용과 비슷합니다. 데몬은 이 비용을 테이블마다가 아니라 요청마다 한 번 치릅니다.

HTTP 파서

파서 행은 400바이트 브라우저 요청을 파싱하고 쿠키 하나를 조회합니다. 132.5 ns가 걸립니다. 크레이트 자체의 타이밍 테스트도 일치합니다(`crates/http`: 같은 머신에서 약 135~180 ns). 파서는 복사하지 않습니다(zero-copy). 모든 필드는 연결 버퍼의 슬라이스입니다.

정책 엔진

브라우저 요청의 전체 분류는 790.8 ns, Gate 프로파일은 75.9 ns가 걸립니다. 이 행들은 요청의 파싱된 클라이언트 주소를 비워 두므로, 엔진은 평가 중에 클라이언트 주소 문자열을 파싱합니다. 데몬은 평가 전에 요청마다 주소를 한 번 파싱합니다. 따라서 이 행에는 데몬이 엔진 밖에서 치르는 작은 비용이 포함됩니다. 엔진의 힙 테이블은 모두 209,052바이트입니다(SID 0002).

WAF 검사

8 KiB 본문의 의미 기반 스캔은 12,784.3 ns, 즉 바이트당 $12784.3/8192 \approx 1.56$ ns가 걸립니다. 두 번째 행은 SQL 인젝션 범주만 감사 모드로, 나머지는 강제 모드로 두고 같은 본문을 스캔합니다. 15,485.4 ns가 걸립니다. 어떤 범주가 강제 대신 감사 모드이면 각 필드는 범주별로 검사됩니다(SID 0004, "inspection modes"). 엔진은 시그니처 오토마톤으로 필드를 한 번 스캔해 시그니처가 나타난 범주 집합을 구한 다음, 그중에서 필요한 범주만 검색합니다. 그래서 한 범주를 감사 모드로 두는 비용은 모든 범주를 강제할 때보다 약 5분의 1 더 들 뿐, 범주마다 스캔 한 번이 드는 것은 아닙니다.

전체 제품 비교

> **학습 목표** 하나의 완전한 프로세스로 실행되는 데몬의 처리량, 지연, CPU 시간, 메모리를 읽고, 운영 구성과 모드를 비교하고, 어떤 행에서 데몬이 한계를 정하고 어떤 행에서 부하 생성기가 한계를 정하는지 말할 수 있어야 합니다.

표는 각 운영 구성과 워크로드에 대해 5초 `wrk` 실행 세 번의 중앙값을 보여 줍니다. 각 워크로드 옆의 상태는 하니스가 기대했고 실제로 받은 응답입니다.

Forward-Auth 모드

인그레스는 요청을 허용할지 데몬에 묻습니다. 원본 서버도 본문 중계도 없으므로 요청 허용 비용을 가장 순수하게 측정합니다.

표 13 forward-auth 모드, Gate와 Shield; 지연 단위는 마이크로초 (`benchmarks/results/product-latest.md`, 생성 2026-10-15T23:15:24Z)

운영 구성	워크로드(상태)	req/s	p50	p90	p99	CPU us/요청	최대 RSS KiB
Gate	`admitted` (200)	211,345	151	175	305	11.84	15912
Gate	`challenged` (401)	198,315	160	180	326	13.15	16176
Gate	`allowed_static` (200)	211,053	152	175	305	11.73	16176
Gate	`attack` (200)	211,618	151	176	305	11.84	16176
Shield	`admitted` (200)	208,101	152	178	308	12.87	15928
Shield	`challenged` (401)	197,532	161	183	327	13.77	16192
Shield	`allowed_static` (200)	211,636	151	175	305	11.78	16192

운영 구성	워크로드(상태)	req/s	p50	p90	p99	CPU us/요청	최대 RSS KiB
Shield	attack (403)	210,549	152	175	307	12.32	16192

forward-auth 모드에서는 데몬이 병목이 아닙니다. 사용 코어 수(CPU 시간을 경과 시간으로 나눈 값, JSON 파일의 cores_busy_median)는 데몬 CPU 4개 중 2.47에서 2.72 사이에 머물고, 모든 행이 초당 197,532에서 211,636 요청 사이에 있습니다. 즉 상한을 정하는 것은 부하 생성기입니다. 그 상한 근처에서는 행 사이의 몇 퍼센트 차이가 실행마다 달라집니다. Gate admitted 행 하나만 해도 세 번의 실행에서 초당 205,151에서 211,513 요청에 걸쳐 있습니다. 이 모드에서 더 유용한 수치는 요청당 CPU 시간입니다. 11.73에서 13.77 us이며, Gate보다 Shield가 조금 더 많고, 챌린지 응답을 렌더링하는 challenged 행이 가장 많습니다.

리버스 프록시 모드

데몬은 스텁 원본 서버 앞에 놓여 허용된 요청과 응답을 중계합니다. 따라서 프록시된 행에는 원본 서버 자체의 비용과 루프백 홉 하나가 더 포함됩니다.

표 14 리버스 프록시 모드, Gate와 Shield; 지연 단위는 마이크로초 (benchmarks/results/product-latest.md, 생성 2026-10-15T23:15:24Z)

운영 구성	워크로드(상태)	req/s	p50	p90	p99	CPU us/요청	최대 RSS KiB
Gate	admitted (200)	142,498	427	548	768	27.68	15896
Gate	challenged (200)	199,465	160	181	324	12.98	16160
Gate	allowed_static (200)	147,303	411	541	759	26.88	16324
Gate	attack (200)	143,900	422	555	745	27.67	16348
Shield	admitted (200)	138,965	438	571	857	28.57	16064
Shield	challenged (200)	195,188	162	185	328	13.73	16328
Shield	allowed_static (200)	145,673	415	551	746	27.27	16328
Shield	attack (403)	209,507	151	177	306	12.17	16328

여기서 프록시된 행(admitted, allowed_static, 그리고 Gate가 중계하는 Gate의 attack)은 CPU가 병목입니다. 사용 코어 수가 4개 중 3.96에서 3.98입니다. 이 행들은 요청당 CPU 26.88에서 28.57 us로 초당 138,965에서 147,303 요청을 처리합니다. 데몬이 직접 답하는 응답의 약 두 배 비용입니다. challenged 행과 Shield의 attack 행은 데몬이 직접 답하므로 forward-auth 행과 비슷하게 보입니다. 포화 상태에서 wrk의 p99는 데몬 자신의 처리 속도에서 나타나는 스케줄러의 꼬리입니다. 그래서 꼬리 지연은 다음 장의 측정값으로 판단합니다.

리버스 프록시 행 세 개에서 wrk는 나오지 않아야 할 2xx, 3xx 이외의 응답을 셌습니다. 세 실행을 합해 Gate attack에서 2,162,579 요청 중 3개, Shield admitted에서 2,082,640 중 1개, Shield allowed_static에서 2,204,358 중 1개입니다. 파일에는 이것이 어떤 상태 코드였는지 기록되지 않았습니다. 데몬 지표를 켜고 실행을 재현하자 sibuna_overloaded_total에 집계된 503 응답이 보였습니다. wrk는 연결 64개를 주소 하나에서 엽니다. 이는 기본 --max-connections-per-ip와 정확히 같습니다. 연결이 4,096번째 요청 뒤에 닫혔을 때, 이전 연결이 자리를 반납하기 전에 wrk가 다시 연

결했습니다. 이제 데몬은 닫는 응답을 쓰기 전에 자리를 반납합니다(SID 0002, "bug fix"). 표는 이 수정 이전의 것입니다.

자원 사용량

표 15 데몬의 크기와 상주 메모리 (benchmarks/results/product-latest.json, 생성 2026-10-15T23:15:24Z, 그리고 primitives-latest.json, 생성 2026-10-15T22:34:59Z)

항목	값
데몬 바이너리	49,113,856바이트
유휴 상주 메모리, 전체 제품 실행	5,592~5,788 KiB
워크로드 중 최대 상주 메모리	15,896~16,348 KiB
워크로드 이후 상주 메모리	16,176~16,348 KiB
유휴 상주 메모리, 기본 연산 실행	6,052 KiB
정책 엔진 테이블	209,052바이트(힙)
봇 시그니처 테이블	42,244바이트(힙)
WAF 테이블	164,908바이트(힙)
브라우저 솔버 모듈(WebAssembly)	14,744바이트

범위는 전체 제품 실행 네 번(모드 두 가지, 운영 구성 두 가지)에 걸친 값이며, 최댓값은 그 16개 워크로드에 걸친 값입니다. 테이블 크기는 엔진이 보고한 힙 크기입니다. 기록은 바이너리 크기를 세부 항목으로 나누지 않습니다.

같은 제공 부하에서의 지연

> **학습 목표** 고정된 제공 부하에서 지연 분포를 읽고, 꼬리 백분위를 그 변동 범위와 함께 읽고, 꼬리 결함 하나를 측정부터 수정까지 따라갈 수 있어야 합니다.

두 표는 모두 `cargo xtask bench latency` 실행 한 번에서 나왔습니다. 오류 열은 생성기의 연결, I/O, 프레이밍 오류를 셉니다.

표 16 forward-auth 모드, Shield, 고정된 제공 요청률에서 허용된 요청; 지연 단위는 마이크로초 (benchmarks/results/latency-latest.md, 생성 2026-10-15T23:07:14Z)

제공 req/s	달성 req/s	p50	p90	p99 (최소-최대)	p99.9	최대	오류	CPU us/요청
10,000	10,000	30	32	37 (36-39)	58	355	0	16.92
25,000	25,000	28	31	34 (32-34)	45	1035	0	15.83
50,000	50,000	25	27	30 (30-30)	40	3029	0	13.83

제공 req/s	달성 req/s	p50	p90	p99 (최소-최대)	p99.9	최대	오류	CPU us/요청
100,000	99,998	24	26	41 (40-42)	62	2358	0	13.48
150,000	149,996	25	36	51 (51-52)	74	1753	0	14.16

표 17 리버스 프록시 모드, Shield, 고정된 제공 요청률에서 허용된 요청; 지연 단위는 마이크로초 (benchmarks/results/latency-latest.md, 생성 2026-10-15T23:07:14Z)

제공 req/s	달성 req/s	p50	p90	p99 (최소-최대)	p99.9	최대	오류	CPU us/요청
10,000	10,000	56	61	65 (65-66)	76	555	0	32.25
25,000	25,000	51	54	58 (58-60)	72	1105	0	28.63
50,000	50,000	47	50	60 (58-66)	74	3066	0	27.45
100,000	99,996	55	75	100 (100-104)	150	2239	0	28.42
150,000	134,344	583934	154416	01906273 (1631234-2218760)	1967134	1989964	4	29.22

forward-auth 모드에서 p50은 모든 요청률에서 24에서 30 us 사이에 머뭅니다. p99는 초당 150,000 요청까지 51 us 이하, p99.9는 74 us 이하입니다. 데몬은 요청당 CPU 13.48에서 16.92 us를 씁니다.

리버스 프록시 모드에서 p50은 초당 100,000 요청까지 47에서 56 us이고, 100,000에서 p99는 100 us, p99.9는 150 us입니다. 초당 150,000 요청에서는 데몬이 포화됩니다. 초당 134,344 요청을 완료하고 지연은 초 단위로 늘어납니다. 개방 루프 테스트라면 용량을 넘는 지점에서 마땅히 이렇게 보여야 합니다. 이 속도는 앞 장의 폐쇄 루프 프록시 처리량(Shield `admitted`의 초당 138,965 요청)과 가깝습니다. 그 행의 오류 네 개는 과부하 중에 발생했습니다.

포화 이전에는 p99의 변동 범위가 좁습니다. 가장 넓은 것은 초당 50,000 요청에서 리버스 프록시의 p99로, 58에서 66 us입니다. 최대 열은 한 실행의 최대 150만 요청 가운데 요청 하나의 값입니다. 공유 호스트에서는 이를 데몬의 성질이 아니라 호스트의 가장 나쁜 순간으로 읽습니다.

측정이 바꾼 설계

외부 부하는 Sibuna의 설계를 바꾸었습니다. 데몬은 각 연결을 전용 스레드에서 처리합니다(SID 0002, SID 0010). 첫 지연 실행에서는 `wrk`가 보여 주지 않았던 꼬리가 나타났습니다. 데몬이 연결 64개를 각각 요청 4,096개 뒤에 닫는 조건에서, 초당 100,000 허용 요청의 99.9번째 백분위는 0.3에서 2 ms였습니다.

SID 0002와 `cmd/sibuna/src/tasks.rs` 문서에 기록된 진단은 스레드 교체입니다. 스레드를 만들고 끝낼 때마다 시그널 스택을 매핑하고 해제하며 스택 페이지를 반환합니다. 이런 주소 공간 변경이 데몬의 다른 스레드를 멈추게 했습니다. 실험 하나로 결론이 났습니다. 연결을 전혀 닫지 않아 실행 중에 스레드가 하나도 만들어지지 않는 빌드는 같은 요청률에서 54에서 60 us를 보였습니다. 수정 방법은 연결 스레드 재사용입니다. 연결을 끝낸 스레드는 대기 상태로 들어갔다가 다음 연결을 맡습니다. 그래서 새 연결이 꾸준히 들어와도 스레드는 만들어지지 않습니다. 10초

동안 대기한 스레드는 끝나며, 다음 스레드를 띄울 때나 다음으로 작업을 마친 스레드가 이를 join합니다. 그래서 버스트가 지나간 뒤 스택이 상주 메모리에 남지 않습니다. 연결 처리 중 패닉이 난 스레드는 대기하지 않고 끝납니다. 이전과 마찬가지로 동시에 처리하는 연결은 최대 `--max-connections`개입니다.

위 표는 수정 후의 실행에서 나왔습니다. 초당 100,000 요청에서 p99.9는 forward-auth 모드에서 62 us, 리버스 프록시 모드에서 150 us입니다. 수정 전 실행은 결과 파일로 남아 있지 않습니다. 그래서 그 수치는 SID 0002와 이 절에만 나옵니다.

풀이 예제: 멈춤이 p99.9에서 드러난 이유 연결 64개로 초당 100,000 요청을 처리하면 연결 하나가 초당 $100000/64 = 1562.5$ 요청을 나릅니다. 데몬은 keep-alive 연결을 요청 4,096개 뒤에 닫습니다(`cmd/sibuna/src/connection.rs`의 `MAX_REQUESTS_PER_CONNECTION`). 따라서 연결 하나는 $4096/1562.5 \approx 2.62$ s 동안 유지되고, 생성기는 초당 약 $64/2.62 \approx 24.4$개의 새 연결을 엽니다. 연결당 스레드 하나라면 매초 스레드 시작 약 24번과 스레드 종료 약 24번이 일어납니다.

이 속도로 10초를 실행하면 요청 1,000,000개를 측정합니다. 그중 1,000개, 즉 초당 100개가 p99.9를 넘고, 초당 1,000개가 p99를 넘습니다. 스레드 교체 한 번이 다른 스레드에서 처리 중인 요청 약 $100/24.4 \approx 4$개를 1밀리초 미만만큼 늦춘다면, 늦어진 요청만으로 상위 0.1%가 채워집니다. 반면 상위 1%는 여전히 평범한 큐 대기로 이루어집니다. 드물고 주기적인 멈춤은 p99를 움직이기 훨씬 전에 먼 꼬리를 움직입니다. 그래서 이런 결함을 보려면 p99.9와 모든 요청에 대한 타이머가 필요합니다.

Sibuna와 Anubis

학습 목표 같은 부하에서 이미 자리 잡은 제품의 수치와 Sibuna의 수치를 나란히 읽고, 어떤 차이가 제품에서 오고 어떤 차이가 설정에서 오는지 말할 수 있어야 합니다.

Techaro의 Anubis는 널리 배포된 작업 증명 게이트로, Sibuna의 Gate처럼 웹 사이트 앞에 놓입니다. 우리는 Anubis v1.27.0의 공식 릴리스 바이너리(`anubis-1.27.0-linux-amd64`, 40,001,698바이트)를 Sibuna와 같은 호스트, 같은 CPU, 같은 워크로드에서 측정했습니다. 두 결과 파일은 다음 명령으로 기록했습니다.

```
cargo xtask bench product --anubis <path/to/anubis>
cargo xtask bench admission --anubis <path/to/anubis>
```

Anubis는 이 프로젝트의 이전 Python 드라이버와 같은 방식으로 실행했습니다. `--bind`, `--metrics-bind`, `--target=` (forward-auth 모드에서는 비워 두고, 리버스 프록시 모드에서는 원본 서버), `--difficulty 2`, `--slog-level ERROR`, `--cookie-secure=false`, 그리고 DNSBL과 허니팟을 끄고 모든 `Mozilla` User-Agent에 챌린지를 제시하는 정책입니다. `GOMAXPROCS`는 Sibuna의 워커 수와 같고, 두 제품의 모든 스레드를 같은 CPU 집합에 고정합니다. Hashcash 작업량도 맞춥니다. Sibuna는 0비트 8개, Anubis는 16진수 0 두 자리로, 기대 해시 수는 똑같이 256입니다. 하니스는 브라우저처럼 각 제품의 세션을 얻습니다. Anubis의 경우 챌린지 페이지를 가져와 `<script id="anubis_challenge">`에서 챌린지를 읽고, `SHA-256(randomData || nonce)`가 `00`으로 시작하는 nonce를 찾아, 페이지의 검증

쿠키와 함께 pass-challenge에 넘깁니다. forward-auth 모드에서 admitted와 challenged 워크로드는 Anubis의 확인 엔드포인트(/.within.website/x/cmd/anubis/api/check)에 묻습니다.

전체 제품 워크로드

네 가지 워크로드, 부하, 통계는 전체 제품 하니스와 같고, 한 가지만 다릅니다. 원본 서버는 이전 드라이버가 썼던 것처럼 부하 CPU에 고정한 caddy respond입니다. 하니스는 매 실행 전에 TIME_WAIT 상태의 소켓이 3,000개 미만으로 줄 때까지 기다립니다. Anubis 리버스 프록시 실행이 수만 개를 남기기 때문입니다.

표 18 forward-auth 모드: 5초 wrk 실행 세 번의 중앙값 (benchmarks/results/comparison-latest.md, 생성 2026-10-16T00:55:19Z).

워크로드	수치	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
admitted	req/s (상태)	210,682 (200)	208,635 (200)	36,509 (200)
	p99 us	305	308	7,477
	CPU us/요청	12.09	13.21	106.30
	최대 RSS KiB	16,228	16,316	32,624
challenged	req/s (상태)	198,251 (401)	193,087 (401)	113,170 (401)
	p99 us	326	342	3,781
	CPU us/요청	13.49	15.02	32.96
	최대 RSS KiB	16,240	16,328	32,696
allowed_static	req/s (상태)	212,569 (200)	212,444 (200)	89,904 (200)
	p99 us	304	304	4,547
	CPU us/요청	11.73	11.95	41.63
	최대 RSS KiB	16,240	16,328	33,396
attack	req/s (상태)	212,086 (200)	211,347 (403)	36,853 (200)
	p99 us	305	306	7,283
	CPU us/요청	12.07	12.56	104.88
	최대 RSS KiB	16,240	16,328	33,764

표 19 리버스 프록시 모드, 원본 서버는 caddy respond: 5초 wrk 실행 세 번의 중앙값 (benchmarks/results/comparison-latest.md, 생성 2026-10-16T00:55:19Z).

워크로드	수치	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
admitted	req/s (상태)	106,376 (200)	103,298 (200)	19,398 (200)
	p99 us	3,331	3,321	11,062

워크로드	수치	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
	CPU us/요청	27.02	27.92	190.40
	최대 RSS KiB	16,528	16,444	39,980
challenged	req/s (상태)	198,402 (200)	197,327 (200)	28,842 (200)
	p99 us	325	328	9,546
	CPU us/요청	13.05	13.78	137.73
	최대 RSS KiB	16,544	16,464	516,872
allowed_static	req/s (상태)	105,427 (200)	106,819 (200)	28,862 (200)
	p99 us	3,320	3,316	30,124
	CPU us/요청	26.24	26.52	135.41
	최대 RSS KiB	16,544	16,464	616,548
attack	req/s (상태)	105,616 (200)	210,483 (403)	23,399 (200)
	p99 us	3,327	306	14,124
	CPU us/요청	26.98	12.28	169.04
	최대 RSS KiB	16,544	16,464	599,516

forward-auth 모드에서 Sibuna는 p99 304에서 342 us로 초당 193,087에서 212,569 요청을 처리했고, Anubis는 p99 3,781에서 7,477 us로 36,509(admitted)에서 113,170(challenged) 요청을 처리했습니다. Sibuna의 사용 코어 수는 4개 중 2.49에서 2.90에 머물렀으므로 부하 생성기가 한계를 정했습니다. Anubis는 3.73에서 3.88이었으므로 CPU가 병목이었습니다. 가장 분명한 수치는 요청당 CPU 시간입니다. 허용된 요청 하나에 Sibuna Gate는 12.09 us, Anubis는 106.30 us를 썼습니다. 이 차이의 일부는 세션 토큰 때문일 가능성이 큽니다. Anubis는 기본적으로 세션에 Ed25519로 서명하고, Sibuna는 키 기반 BLAKE3 MAC을 확인합니다(기본 연산 표에서 155.8 ns). 아래 요청 허용 측정에서 HS512 토큰을 쓰면 Anubis의 세션 확인이 절반 넘게 늘어나는데, 이는 이 해석을 뒷받침합니다.

리버스 프록시 모드에서 Sibuna의 프록시된 행은 사용 코어 수 2.77에서 2.98로 초당 103,298에서 106,819 요청을 처리했습니다. 이는 product-latest.json에서 스텁 원본 서버로 얻은 138,965에서 147,303보다 낮고, 데몬은 CPU가 병목이 아니므로 한계는 데몬 밖에 있습니다. 원본 서버인 caddy respond가 wrk와 부하 CPU를 공유합니다. 그 꼬리는 2.1에서 2.4 ms의 p90에서 드러납니다. Anubis는 사용 코어 수 3.68에서 3.97, 요청당 CPU 135에서 190 us로 초당 19,398에서 28,862 요청을 처리했습니다. Anubis의 상주 메모리는 admitted 워크로드의 39,980 KiB에서 challenged의 516,872 KiB, allowed_static의 616,548 KiB로 늘었고, 워크로드가 끝난 뒤에도 589,276 KiB를 유지했습니다(유휴 시 21,552 KiB). 증가는 모든 요청이 고유 id를 가진 새 챌린지를 받는 challenged에서 시작되었습니다. 발급한 챌린지마다 기록을 보관한다고 짐작할 수 있지만, 하니스는 원인을 확인하지 않았습니다. Sibuna의 최댓값은 두 모드의 모든 행에서 16,228에서 16,544 KiB 사이에 머물렀습니다.

몇몇 행은 같은 조건의 비교도 아닙니다.

- forward-auth 모드에서 Anubis는 `/robots.txt`와 `attack` 경로에 원본 서버 없이 직접 `200`으로 답합니다. Sibuna는 이 모드에서 다른 모든 요청과 같은 방식으로 답합니다.
- Anubis에는 요청 검사가 없으므로 `attack` 워크로드의 SQL 인젝션을 허용합니다. Shield는 더 많은 작업을 한 뒤 `403`으로 이를 거부합니다. Gate는 Anubis처럼 이를 허용합니다.
- 두 제품은 세션 서명 방식이 다르며(Anubis는 기본적으로 Ed25519, Sibuna는 키 기반 BLAKE3 MAC), 서로 다른 챌린지 페이지를 렌더링합니다.

요청 허용 연산

요청 허용 하니스는 forward-auth 모드에서 요청 허용을 이루는 네 가지 연산의 시간을 잽니다. 각 제품은 CPU 두 개에서 실행합니다(Sibuna 워커 2개, `GOMAXPROCS=2`). 부하 CPU에 있는 클라이언트 스레드 두 개가 각각 keep-alive 연결 하나로 각 배치를 나누어 처리합니다. 배치는 연산 200개이며, 검증의 경우 배치 전에 준비한 증명 64개입니다. 배치 7개의 중앙값을 보고합니다.

- 세션 확인: 유효한 세션으로 확인 엔드포인트에 요청합니다(`200`).
- 미인증 확인: 같은 요청을 쿠키 없이 보냅니다(`401`).
- 챌린지 초기화: 브라우저가 챌린지를 시작하려고 가져오는 것입니다. Sibuna는 중간 페이지와 챌린지 레코드의 두 요청이고, Anubis는 챌린지가 포함된 페이지 하나입니다.
- 증명 검증: 연산마다 새 증명 하나를 검증합니다. Sibuna의 증명 챌린지는 25 ms 간격으로 발급됩니다. 이는 적응형 난이도가 비트를 더하기 시작하는 초당 50개보다 낮으므로, 맞춘 작업량이 유지됩니다. Sibuna는 `--challenge-rate-limit 100000000`으로 실행하므로, 제한기는 검사되지만 소진되지는 않습니다.

Anubis는 기본 Ed25519 토큰으로 한 번, `--hs512-secret`으로 한 번, 두 번 실행했습니다.

표 20 초당 요청 허용 연산 수, 클라이언트 2개, 제품은 CPU 2개 (`benchmarks/results/admission-latest.md`, 생성 2026-10-16T00:55:44Z)

제품(토큰)	세션 확인	미인증 확인	챌린지 초기화	증명 검증	이후 RSS, KiB
Sibuna(키 기반 BLAKE3)	88,104	78,997	42,664	57,924	7,792
Anubis 1.27.0(Ed25519)	10,951	28,619	7,893	12,668	33,400
Anubis 1.27.0(HS512)	16,976	27,785	7,952	19,261	29,840

배치 7개의 범위는 파일에 있습니다. Sibuna의 범위는 중앙값의 11% 이내에 있습니다. Anubis의 범위는 더 넓습니다(Ed25519 토큰을 쓴 미인증 확인은 25,394에서 32,741에 걸칩니다). 또 연산 200개짜리 배치는 2에서 25 ms밖에 걸리지 않으므로, 이 수치는 10% 정도의 정밀도로만 읽습니다. 느린 클라이언트의 연산당 시간이 왕복 시간을 보여 줍니다. Sibuna 세션 확인은 22.7 us, Anubis 확인은 Ed25519 토큰에서 182.5 us, HS512에서 117.7 us입니다.

위의 어떤 행에서도 Anubis가 Sibuna보다 더 많은 요청을 처리하거나 CPU나 메모리를 덜 쓰지 않았습니다. 이는 이 호스트, 이 워크로드에 대한 진술일 뿐 그 이상은 아닙니다.

Anubis 행 읽기 Anubis는 기능이 충실하고 널리 배포된 제품이며, 이 행들은 Anubis가 느리다는 주장이 아닙니다. Anubis는 가비지 컬렉션 런타임에서 실행되고, 검증에 공유 비밀이 필요 없도록 기본적으로 세션에 Ed25519로 서명하며, 여러 챌린지 방식과 풍부한 정책 언어를 제공하고, 챌린지 상태를 저장소에 보관합니다. 모두 이 하니스가 측정하지 않는 이점이 있는 설계 선택입니다. 이 행들은 2026년 10월, 한 호스트에서 한 가지 부하 형태로 각 제품이 요청 하나에 치른 비용을 보여 줍니다. Sibuna는 새로운 제품이며, 이 수치는 이미 자리 잡은 제품과 나란히 놓고 측정한 첫 결과입니다.

콘솔 오버헤드

SID 0007은 콘솔의 격리 계약을 정합니다. 콘솔을 켰을 때의 비용은 데이터 플레인 처리량의 최대 1%, p99 지연의 최대 10%로 제한됩니다. 이 비용은 구성 행렬(컴파일에서 제외, 컴파일했지만 비활성, 유휴, 활성 대시보드 8개)과 워크로드에 대해, 번갈아 반복한 실행과 보고된 불확실성으로 측정합니다. 결론이 나지 않거나 잡음이 큰 실행은 준수를 입증하지 못하며, 기본 연산 기준값과 스모크 실행은 이 관문을 대신할 수 없습니다.

SID 0010은 측정 하나를 기록합니다. 콘솔을 컴파일해 포함한 상태에서 비활성, 저장소만, 또는 활성이지만 유휴일 때, 16개 전체 제품 워크로드는 콘솔 없이 빌드한 데몬과 약 1% 이내로 차이 나며, 이는 잡음 범위 안입니다. 기록은 이에 대한 결과 파일을 밝히지 않습니다. 부하 상태에서 활성 대시보드 8개를 띄운 구성은 측정하지 않았으므로, SID 0007의 공식 관문은 아직 미해결로 남아 있습니다.

대시보드 8개 실행은 오버헤드가 아니라 기능을 확인합니다. `sibuna-browser dashboards`는 콘솔, 워커 4개, 그리고 `/static/*`, `/about`, `/missing/*`, `/broken/*`는 허용하고 나머지에는 모두 챌린지를 제시하는 정책을 갖춘 릴리스 데몬을 SQLite 노드 하나에서 시작했습니다. 원본 서버 픽스처는 `/missing/*`에 404로, `/broken/*`에 500으로 답합니다. 생성기는 30초 동안 초당 요청 200개를 보내며 경로 10개(그중 하나는 SQL 인젝션), User-Agent 3개, 전달된 주소 500개를 번갈아 사용했습니다. 각 대시보드는 자기 세션을 가진 별도 Chromium에서 실행했습니다. 대시보드 1부터 3은 관리자 세션으로, 대시보드 4부터 8은 첫 대시보드가 발급한 코드로 연 벽면 디스플레이로 실행했습니다. 콘솔은 주소마다, 계정마다 분당 다섯 번의 로그인만 허용하기 때문입니다.

표 21 노드 하나에서 띄운 트래픽 대시보드 8개 (`benchmarks/results/console-8-dashboards-2026-10-15.md`, 기록 2026-10-15, Chromium 155.0.8059.39). 결과: PASS

항목	값
데이터 플레인 요청	30.0 s 동안 6,000개(초당 200개, 스레드 4개, 요청마다 연결 하나)
응답	6,000: 200 x 1,200, 401 x 3,000, 403 x 600, 404 x 600, 500 x 600
오류(응답 없음)	0
지연 p50 / p90 / p99 / 최대	246 / 311 / 373 / 552 us (연결 시간 포함)
시작 시 데몬 VmRSS	36,152 KiB

항목	값
대시보드 8개를 연 상태의 데몬 VmRSS	38,028 KiB
트래픽 이후 데몬 VmRSS	41,496 KiB (VmHWM 41,496 KiB)

타일은 트래픽 동안 2초마다, 그리고 트래픽이 끝나고 3초 뒤에 읽었습니다. 모든 대시보드가 같은 합계(Requests 6,000, Admitted 2,400, Challenged 3,000, Policy denied 600, Origin 4xx 600, Origin 5xx 600)로 끝났고, stats 변경분을 134에서 139개 받았으며, WebSocket을 계속 열어 두었고, 예외, 콘솔 오류, Content-Security-Policy 위반을 하나도 보고하지 않았습니다. 브라우저 구독자 8개는 콘솔의 한도인 64개보다 훨씬 적습니다. 이 실행의 데몬은 전체 제품 하니스의 메모리 모드 데몬과 달리 SQLite 저장소와 콘솔을 갖추었으므로, 상주 메모리를 그것들과 비교할 수 없습니다.

측정하지 않은 것

표 22 이 부가 측정하지 않는 것과 측정을 대신하는 것

대상	여기서 측정하지 않는 이유	대신하는 것
SafeLine 커뮤니티 에디션 9.x	Docker Compose 스택으로만 배포됩니다. 호스트에 Docker 데몬이 없었고, 탐지기 이미지는 그곳에서 컴파일할 수 있는 오픈 소스 빌드가 아닙니다	공급업체가 발표한 사실(2026년 9월 확인): 탐지율 71.65%(balance)와 76.17%(strict), 오탐 0.07%(balance)와 0.22%(strict); 최소 호스트는 SSSE3를 지원하는 Linux x86_64 또는 arm64, CPU 코어 1개, RAM 1 GB, 디스크 5 GB, Docker 20.10.14+, Compose 2.0+
Cloudflare WAF	호스팅 서비스입니다. 루프백 측정은 WAF가 아니라 네트워크 경로를 측정하게 됩니다	발표된 요금제 할당량(2026년 9월 확인): 사용자 지정 규칙 Free 5, Pro 20, Business 100, Enterprise 1000; 속도 제한 규칙 Free 1, Pro 2, Business 5, Enterprise 100
부하를 받는 클러스터	PostgreSQL 데이터베이스 하나를 쓰는 여러 노드를 구동하는 하니스가 아직 없습니다	SID 0011이 명시하는 프로토콜 속성(공유 시드 하나, 발급자에 묶인 챌린지)과 노드 3개 기능 테스트(기준 2); 측정 없음
요청 경로의 저장소	모든 하니스에서 두 데몬 모두 메모리 모드로 실행합니다	SID 0011: 저장소 스레드 하나가 모든 SQL을 맡고, 요청 경로는 그 스레드를 기다리지 않습니다

대상	여기서 측정하지 않는 이유	대신하는 것
TLS, WAN, 실제 브라우저	하니스는 루프백과 HTTP를 사용합니다. 세션 챌린지는 경우마다 `sibuna-pow`로 한 번 풉니다	없음. 각 배포 환경에서 자기 경로를 직접 측정합니다
브라우저 솔버의 풀이 시간	하니스는 네이티브 솔버로 8비트에서 풉니다	위의 모듈 크기. 제VII부는 네이티브 시간이 브라우저 풀이 시간을 대신할 수 없는 이유를 설명합니다
할당 횟수	하니스는 할당기를 계측하지 않습니다	제V부의 정상 상태 테스트
부하 상태에서 활성 대시보드 8개를 띄운 콘솔 격리	SID 0007의 공식 관문을 아직 실행하지 않았습니다	SID 0010의 수치(약 1% 이내, 콘솔 유휴)와 위의 기능 검사 실행

이 표의 행은 제품 성능에 관한 주장이 아닙니다. SafeLine이 발표한 수치는 처리량이 아니라 탐지 품질에 관한 것이고, Cloudflare는 요청당 비용 대신 요금제 할당량을 발표합니다. Docker 호스트가 있으면 누구나 같은 `wrk` 워크로드로 SafeLine을 측정할 수 있습니다.

연습 문제 8.1. 다른 머신에서 `cargo xtask bench primitives`를 실행하고, Hashcash 검증 행과 토큰 검증 행을 기준 호스트와 비교하세요. 둘 중 어느 쪽이 하드웨어 해시 명령어에 더 크게 의존하며, 그 이유는 무엇입니까?

연습 문제 8.2. forward-auth 표에서 행의 사용 코어 수를 계산하세요. 요청당 CPU 마이크로초에 초당 요청 수를 곱하고 100만으로 나눈 값입니다. 데몬은 CPU가 병목이었습니까? 그다음 요청당 CPU 마이크로초를 첫 번째 표의 분류 비용과 비교하세요. 요청당 CPU 비용은 왜 훨씬 더 높으며, 어떤 구성 요소가 그 차이를 만듭니까?
힌트: `accept`부터 플러시된 응답까지의 과정 가운데 기본 연산 측정 모음이 시간을 재지 않는 것을 모두 나열하세요.

불변식 설명하기. 이 부의 모든 수치는 반복에 대한 중앙값이며, 그 변동 범위를 옆에 적습니다. 배치 중앙값 수치와 요청별 p99.9의 차이를 설명하고, 왜 기본 연산 표는 전자를, 지연 표는 후자를 보고하는지, 그리고 왜 후자만이 위에서 설명한 스레드 교체 멈춤을 드러낼 수 있었는지 설명하세요.

제I X부

운영과 배포

두 가지 운영 구성, 모든 명령줄 플래그, *forward-auth* 구성 예, 정책 파일, *SQLite* 또는 *PostgreSQL*을 쓰는 영속 저장소, 공유 데이터베이스 하나를 쓰는 클러스터, 패키징을 다룹니다.

운영 구성과 설정

학습 목표 이 장을 마치면 Sibuna를 리버스 프록시나 forward-auth 검증기로 실행하고, 운영 구성을 선택하고, 작업 증명 계층과 난이도를 설정하고, 마스터 비밀 값을 관리할 수 있어야 합니다.

운영 구성 선택

- *Gate* (`--gate`, 별칭 `--no-waf`): 작업 증명 요청 허용, 세션, 선언형 규칙, 평판, GCRA 제한, 차단을 제공합니다.
- *Shield* (기본값, `--shield`, 별칭 `--waf`): Gate에 의미 기반 방화벽을 더합니다.
- 분산 배포 (두 운영 구성 모두에서 선택 가능): `--data-dir`는 정책, 평판, 포렌식 기록을 로컬 SQLite 파일에 보관합니다. `--database-url`과 `--cluster-node`를 함께 쓰면 여러 노드가 하나의 PostgreSQL 데이터베이스에 연결됩니다(SID 0011).

명령줄 레퍼런스

이 표는 `sibuna --help`의 모든 옵션을 나열합니다. 범위는 파서가 강제하는 값입니다.

표 23 데몬 옵션

플래그	기본값	의미
`--port, -p`	`8080`	수신 포트, 1에서 65535까지
`--host, -h`	`0.0.0.0`	수신 주소, IP 주소 리터럴
`--upstream-host`	`127.0.0.1`	원본 서버 호스트(리버스 프록시 모드)
`--upstream-port, -u`	`3000`	원본 서버 포트
`--mode, -m`	`reverse_proxy`	`reverse_proxy` 또는 `forward_auth` (`proxy`와 `auth`도 가능), 한 번만 지정
`--workers, -w`	CPU당 하나	수신 소켓을 공유하는 accept 스레드 수, 0(CPU당 하나)에서 64까지. 수락한 연결마다 전용 스레드를 하나씩 둡니다

플래그	기본값	의미
`--max-connections`	`1024`	동시에 처리하는 연결 수, 1에서 8192까지. 그 이상은 `503`으로 응답합니다
`--max-connections-per-ip`	`64`	소켓 피어 주소 하나가 유지할 수 있는 연결 수, 0(제한 없음)에서 8192까지. IPv6 피어는 `--ipv6-prefix` 네트워크 단위로 세며, 전달된 헤더는 관여하지 않습니다. 그 이상은 `503`으로 응답합니다
`--idle-timeout`	`15`	가장 긴 무활동 시간(초), 0에서 86400까지. 요청 헤드는 이 시간 안에 도착해야 합니다. 원본 서버 응답은 읽을 때마다 이 시간을 갱신하므로, 느린 스트림은 바이트가 흐르는 동안 유지되고 아무것도 보내지 않는 원본 서버는 양쪽 소켓 모두 연결이 끊깁니다. 프록시를 거치는 업로드는 주기마다 16 KiB를 전달해야 합니다(느린 본문 공격에 대한 최소 속도). 클라이언트 소켓에도 같은 길이의 쓰기 타임아웃이 걸리므로, 그 시간 동안 한 바이트도 받지 않는 클라이언트는 연결을 잃습니다
`--websocket-idle-timeout`	`300`	업그레이드된 연결의 유휴 기한(초), 0에서 86400까지
`--trust-forwarded`	꺼짐. forward-auth에서는 켜짐	피어가 보낸 `X-Forwarded-For` / `X-Real-IP`를 신뢰합니다
`--no-trust-forwarded`	꺼짐	forward-auth 모드에서도 이 헤더를 신뢰하지 않습니다
`--trusted-hops`	`1`	전달된 신원을 신뢰할 때, 오른쪽에서 이 수만큼 떨어진 위치의 `X-Forwarded-For` 항목을 클라이언트로 봅니다. 1에서 8까지
`--ja4-header`	꺼짐	신뢰할 수 있는 인그레스가 클라이언트의 JA4 TLS 지문을 보내는 요청 헤더(예: `cf-ja4`). 토큰 문자 1개에서 64개이며, `X-Forwarded-*`, `X-Real-IP`, `X-Sibuna-*`, `Forwarded`는 쓸 수 없습니다. 전달된 신원을 신뢰할 때만 읽습니다("인그레스에서 받는 JA4 지문" 참조)
`--ipv6-prefix`	`64`	속도 제한, 차단, 클라이언트별 연결 상한이 적용되는 IPv6 접두사 길이, 32에서 128까지
`--algorithm, -a`	`posw`	`posw` 또는 `hashcash`

플래그	기본값	의미
`--difficulty, -d`	`16`	작업 비트, 1에서 64까지: Hashcash의 0비트 수 또는 PoSW 깊이에 3을 더한 값
`--posw-challenges`	`16`	순차 작업 증명 하나당 개방 수, 1에서 32까지
`--token-scheme`	`mac`	`mac` (키 기반 BLAKE3) 또는 `ed25519`
`--token-ttl`	`86400`	세션 수명(초), 1에서 31,536,000까지
`--challenge-ttl`	`300`	챌린지 수명(초), 1에서 86,400까지
`--secret-file, -s`	무작위 생성	16진수 문자 64개 또는 원시 바이트 32개. `SIBUNA_SECRET` (16진수 문자 64개)도 사용할 수 있습니다
`--cookie-name`	`__sibuna_token`	세션 쿠키 이름, 토큰 문자 1개에서 64개
`--secure-cookie`	꺼짐	`Secure` 속성을 추가합니다
`--gate` / `--shield`	shield	운영 구성 선택
`--rate-limit`	`100`	클라이언트별 윈도당 요청 수(GCRA 버스트)
`--rate-window`	`10`	윈도(초), 1에서 86400까지
`--challenge-rate-limit`	`30`	클라이언트별 윈도당 챌린지 발급과 검증 횟수, 요청 예산과 별개
`--ban-seconds`	`3600`	허니팟 차단 기간, 0에서 31,536,000까지
`--policy-file, -P`	없음	선언형 JSON 정책
`--rate-table-cells`	`262144`	속도 제한 테이블당 셀 수, 셀마다 16바이트, 4,096에서 16,777,216까지, 2의 거듭제곱으로 올림
`--ban-table-slots`	`65536`	차단 테이블 슬롯 수, 슬롯마다 24바이트, 1,024에서 16,777,216까지, 2의 거듭제곱으로 올림
`--spent-tags`	`1048576`	재전송을 막기 위해 기억하는 해결된 챌린지 수, 항목마다 24바이트, 65,536에서 16,777,216까지
`--data-dir, -D`	없음	단일 노드: SQLite 파일 `<path>/sibuna.db`에 영속 저장합니다
`--database-url`	없음	클러스터: 공유 PostgreSQL 데이터베이스 (`postgres://` 또는 `postgresql://`)
`--database-url-file`	없음	소유자만 읽을 수 있는 파일에서 데이터베이스 URL을 읽습니다

플래그	기본값	의미
`--cluster-node`	없음	이 노드의 id. 챌린지 키에 묶이고 인시던트 식별자의 상위 비트가 됩니다. 데이터베이스 URL을 쓸 때 반드시 지정해야 합니다
`--storage-poll-ms`	`500`	저장소 틱 간격, 1에서 60,000 ms까지
`--verbose, -v`	꺼짐	상세 로그 출력
`--help / --version`		도움말(콘솔이 컴파일되어 있으면 콘솔 옵션 먼저) 또는 버전을 출력하고 종료합니다

`--data-dir`와 데이터베이스 URL은 함께 쓸 수 없고, `--database-url`과 `--database-url-file`도 함께 쓸 수 없습니다. `--cluster-node` 없이 데이터베이스 URL을 지정하면 거부합니다. Zaxonlite 클러스터 옵션 `--cluster-listen`, `--cluster-peer`, `--cluster-secret-file`, `--cluster-tls-cert`, `--cluster-tls-key`, `--cluster-tls-ca`는 제거되었습니다. 이 중 하나를 지정하면 `RemovedOption`과 SID 0011을 가리키는 힌트를 내고 시작을 멈춥니다. 콘솔 옵션은 다음 장에 나와 있습니다.

마스터 비밀 값 `--secret-file`이나 `SIBUNA_SECRET`이 없으면 데몬은 무작위 시드를 뽑고, 배너에 "random (set –secret-file)"을 출력합니다. 그러면 토큰과 챌린지는 프로세스와 함께 사라지고, 노드마다 시드가 다른 클러스터에서는 노드끼리 서로의 쿠키를 거부합니다. `head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret`로 시드를 만들고 권한을 0600으로 설정하세요.

콘솔 미리보기

콘솔은 기본 `console` 기능으로 데몬에 컴파일되지만 `--console`을 지정할 때만 시작합니다. 콘솔에는 저장소가 필요합니다. `--data-dir`이나 `--database-url` 없이 `--console`을 지정하면 `CONSOLE001` (`StorageRequired`)로 거부됩니다. 콘솔을 빼고 컴파일하려면 `cargo build --release -p sibuna --no-default-features --features sqlite`를 사용하세요. `--no-default-features`만 지정한 빌드에는 저장소도 콘솔도 없습니다. 데몬을 중지한 상태에서 로컬로 초기 설정을 합니다:

```
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
```

`sibuna console init-admin`도 같은 명령입니다. `http://127.0.0.1:19446/console/`을 열고 생성된 임시 비밀번호를 바꾸세요. 이 비밀번호는 첫 로그인 때 반드시 바꿔야 하며 한 시간 뒤 만료됩니다. 로그인한 뒤의 탐색 메뉴는 움직이는 국가 지구본, 요청과 챌린지 통계, 인시던트 조사, 정책과 검사 편집기, 사용자, 범위가 정해진 API 토큰, 감사 기록, 서비스 노드 제어를 제공합니다. 인증 화면은 지구본 형상이나 텔레메트리를 불러오지 않습니다. 인터페이스는 데몬의 빌드 스크립트가 `cmd/console-ui` (크레이트 `sibuna-console-ui`)에서 빌드한 WebAssembly 모듈입니다. 이 모듈과 커밋된 스타일시트는 일반 빌드에 포함됩니다.

표 24 콘솔 옵션

플래그	의미
`--console <ip:port>`	이 리스너에서 콘솔을 시작합니다(IPv6는 대괄호 안에, 포트는 0이 아님). `--data-dir` 또는 `--database-url`이 필요합니다
`--console-key-file <path>`	16진수 문자 64개, 소유자 전용 권한. 저장된 2단계 인증 비밀 값과 웹훅 비밀 값을 보호합니다
`--console-origin <https-origin>`	프록시 뒤에서 사용하는 정식 HTTPS 출처
`--console-behind-proxy`	TLS를 종료하는 프록시가 모든 요청을 전달합니다
`--console-trusted-proxy <cidr>`	전달 필드를 실어 보내도 되는 연결이 오는 네트워크. 반복 지정 가능, 최대 16개
`--console-advertise <origin>`	다른 콘솔이 이 콘솔을 가리킬 때 보여 주는 링크
`--console-location <latitude,longitude>`	지구본 위에 선언하는 이 노드의 위치
`--console-probe <node-id>=<http://ip:port>`	상태를 점검할 피어 데이터 평면 리스너. 반복 지정 가능, 최대 8개
`--console-peer <node-id>=<https://origin>`	관리 피어. 반복 지정 가능, 최대 8개
`--console-peer-key-file <path>`	피어 키: 소유자 전용, 16진수 문자 64개, 콘솔 키와 독립적
`--console-peer-ca-file <path>`	피어용 선택적 PEM 신뢰 앵커. 시스템 루트 인증서를 대체합니다
`--console-query-steps <n>`	집계 읽기 한 번당 SQLite 단계 수, 100,000에서 50,000,000까지, 기본값 4,000,000
`--console-capture-heads`	인시던트마다 마스킹한 요청 헤드와 원본 서버 응답 헤드를 저장합니다. 기본값은 꺼짐
`--console-capture-header <name>`	저장된 헤드에 이 헤더의 값을 남깁니다. 반복 지정 가능, 최대 16개. `--console-capture-heads`가 필요합니다
`--console-cookie-secure`	프록시 없이도 콘솔 쿠키에 `Secure` 속성을 붙입니다(파서는 받지만 `--help`에는 나오지 않음)

`--console` 없이 지정한 콘솔 옵션, 반복된 옵션, 형식이 잘못된 값은 `CONSOLE002`와 해당 옵션의 오류 이름을 내고 시작을 멈춥니다. 콘솔 규칙을 어기는 설정(예: 신뢰할 수 있는 프록시 없이 루프백이 아닌 주소에서 수신)은 `CONSOLE001`로 시작을 멈춥니다.

콘솔 둘러보기

아래 화면은 SQLite 저장소를 쓰는 루프백 노드 하나에서 릴리스 빌드로 캡처했습니다. 정책은 기본 챌린지 위에 `ALLOW` 규칙 네 개를 두었고, 약 4분 동안 합성 트래픽을 보냈습니다. 트래픽은

문서용 대역의 주소, SQL 인젝션과 크로스 사이트 스크립팅 탐색, 허니팟 적중 세 번, 헤드리스 Chromium에서 해결한 챌린지 일곱 개로 이루어졌습니다. 따라서 모든 수치는 예시일 뿐입니다. 화면은 `cargo xtask browser console --viewport-only`로 캡처했습니다. 이 명령은 Chromium에서 1440 x 960 픽셀로 콘솔을 순회하며 모든 페이지를 검사합니다. 각 페이지는 제목에 적힌 질문 하나에 답하며, 제품, 서비스 노드, 페이지, 돌아가기 링크를 같은 자리에 표시합니다.

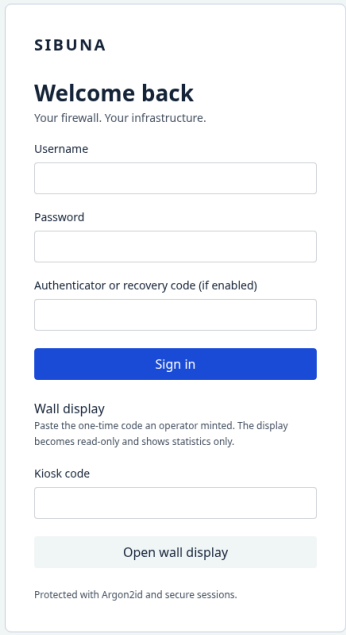


그림 13 로그인. 공개 화면은 지구본 형상, GeoIP 데이터, 텔레메트리를 불러오지 않습니다. 두 번째 양식은 벽면 디스플레이 코드를 읽기 전용 통계 세션으로 교환합니다.

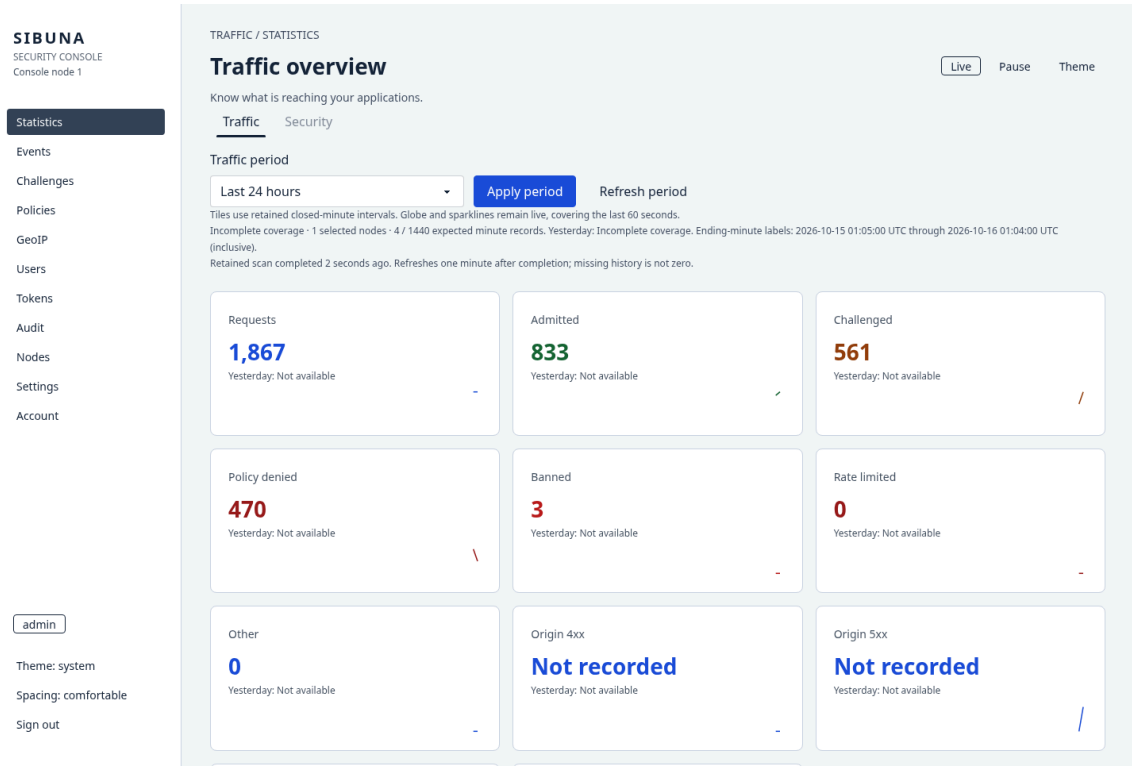


그림 14 통계, 트래픽, 최근 24시간. 기간 선택 아래의 안내문은 범위를 알려 줍니다. 예상되는 분 단위 기록 1,440개 가운데 4개만 있으므로 모든 "어제" 값은 "사용할 수 없음"으로 표시됩니다. 타일은 보존된, 이미 마감된 분의 집계를 실시간 60초 스파크라인과 함께 보여 줍니다. 결정 색상(허용은 녹색, 챌린지는 황색, 거부는 적색)은 모든 패널에서 같습니다. 보존된 기간의 원본 서버 4xx와 5xx는 "기록되지 않음"으로 표시됩니다. 분 단위 기록은 원본 서버 수에 의미를 부여하는 프록시 모드를 보관하지 않기 때문입니다. 이번 부팅 이후의 실시간 합계에는 이 값이 표시됩니다. 보이는 타일 아래에는 활성 차단 항목과 정상 노드를 보여 주는 수준 타일 두 개, 타임라인, 그리고 요청 경로와 참조 호스트의 순위를 매기고 클라이언트 운영 체제, 브라우저, 응답 상태를 세는 표본 패널이 있습니다.

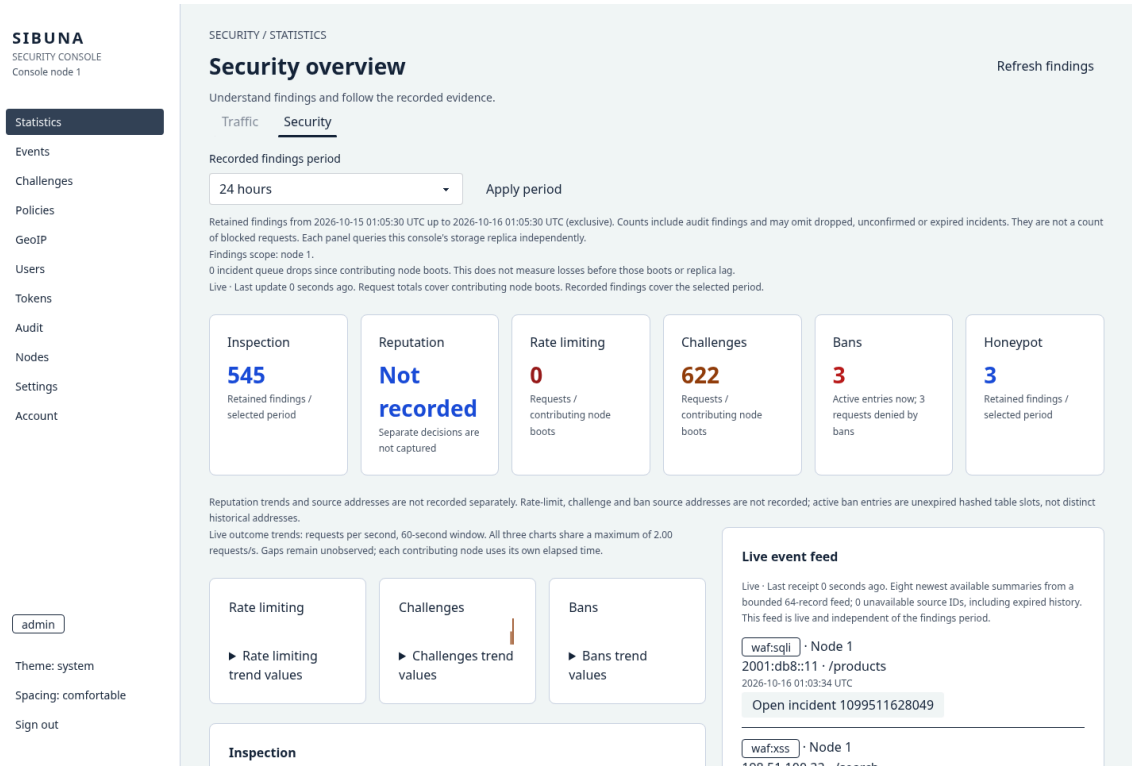


그림 15 통계, 보안. 선택한 기간의 모듈별 타일 하나씩, 실시간 결과 추세, 최신 탐지 결과(여기서는 `waf:sqli`와 `waf:xss`)를 보여 주는 실시간 이벤트 피드가 있습니다. 기록되지 않은 값은 0을 보여 주는 대신 그렇다고 밝힙니다. 평판 타일은 "기록되지 않음"으로 표시됩니다. 페이지 아래쪽에는 같은 크기의 구간 60개에 걸친 탐지 결과, 공격 범주, 공격받은 경로가 있습니다.

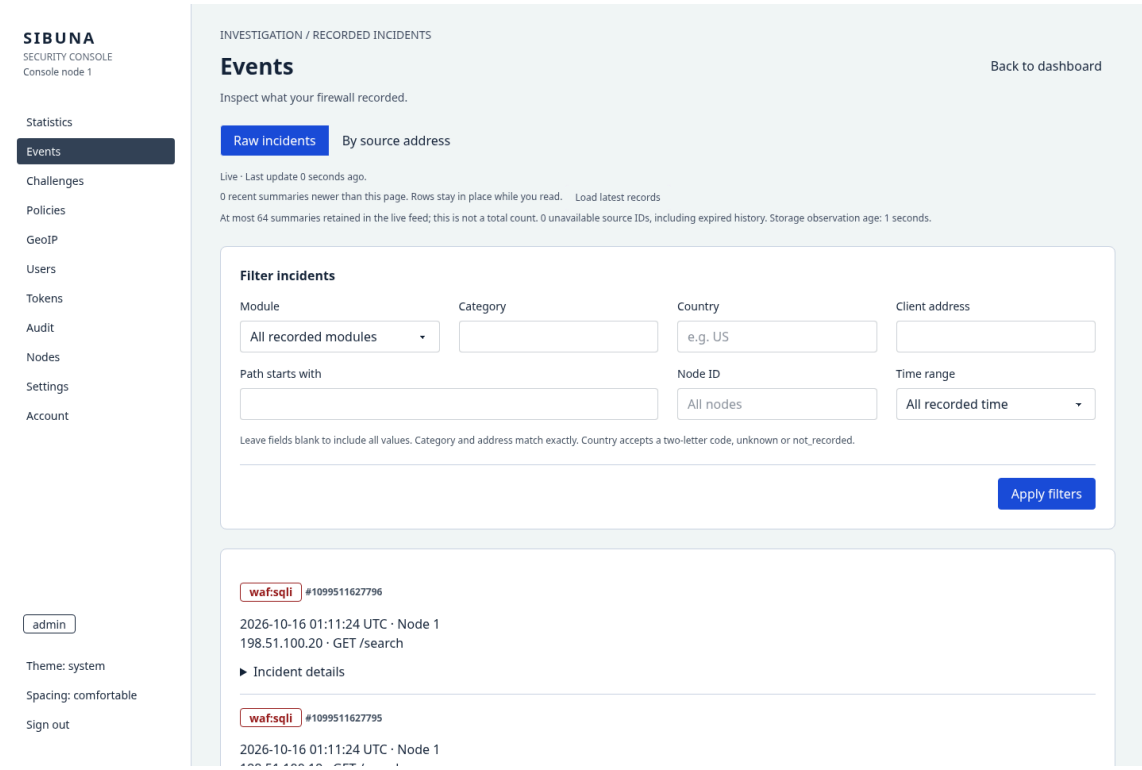


그림 16 이벤트. 기록된 인시던트 위에 필터 양식이 있고, 인시던트는 최신순으로 표시됩니다. 각 인시던트는 그 자리에서 펼쳐져 증거를 보여 줍니다. 선택된 로컬 응답, 바이트 길이, 캠페인 후보, 그리고 일치한 규칙, 점수 항목, JA4 지문에 대한 명시적인 "기록되지 않음" 항목이 있습니다. `--console-capture-heads`를 지정하면 마스킹한 요청 헤드(리버스 프록시를 거쳐 감사 대상으로 허용된 요청이라면 원본 서버 응답 헤드도)를 요청 시 펼쳐 볼 수 있습니다. 헤드는 UTF-8 또는 Latin-1로 표시되며, 응답 조건(캡처됨, 로컬, forward-auth에서 관찰 안 됨, 원본 서버 사용 불가)과 cURL 명령으로 복사하는 기능이 함께 제공됩니다. 목록에 있는 헤더 값만 보관되며, `--console-capture-header <name>`으로 그 목록에 추가합니다. 그렇지 않으면 헤드는 기록되지 않음으로 표시됩니다. 거부와 허용 동작은 해당 주소가 미리 채워진 IP 그룹 양식을 엽니다.

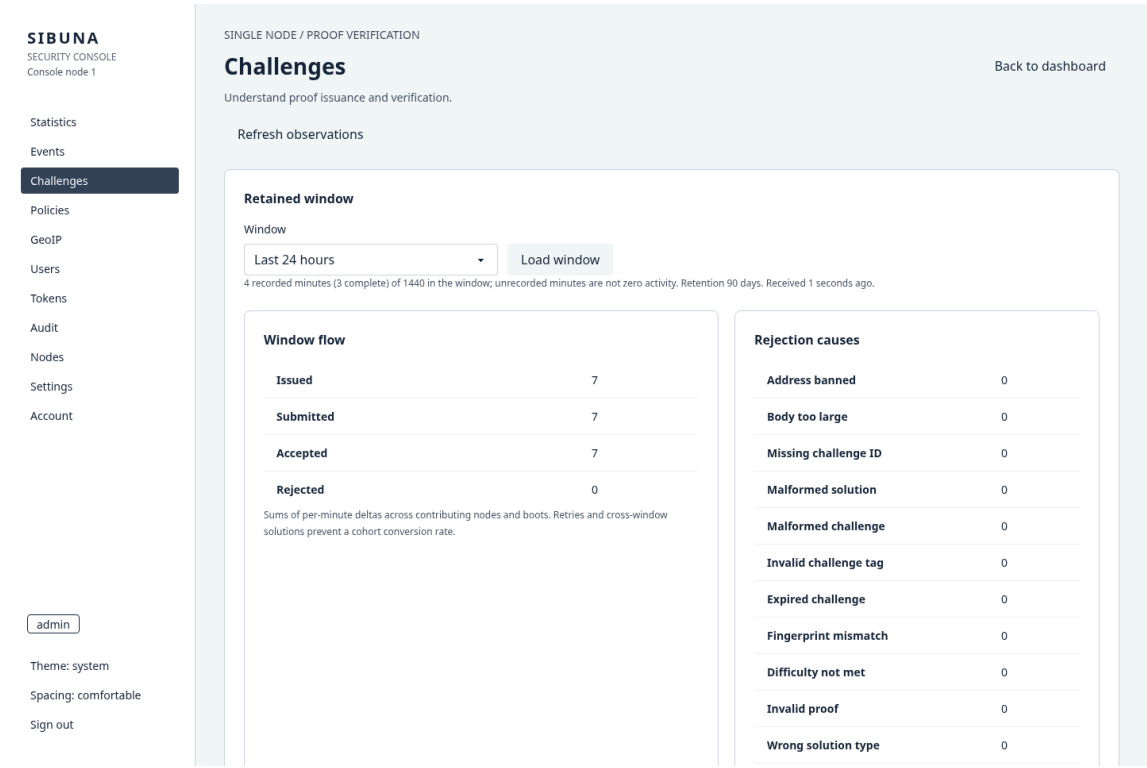


그림 17 챌린지. 보존 기간(1시간, 1일, 7일)은 영속적인 분 단위 기록을 합산하고, 기록되고 완결된 분이 몇 개인지 밝힙니다. 여기서는 챌린지 일곱 개가 발급, 제출, 수락되었고 거부 원인은 모두 0입니다. 페이지 아래쪽에는 이번 부팅 이후의 실시간 합계, 설정된 매개변수와 가장 최근에 발급된 매개변수, 알고리즘과 매개변수 구간별로 나눈 수락된 풀이 시간 히스토그램, 적응형 난이도 전환, 거부와 허용 바로 가기가 있는 주소별 기록이 있으며, 7일 동안 보존됩니다.

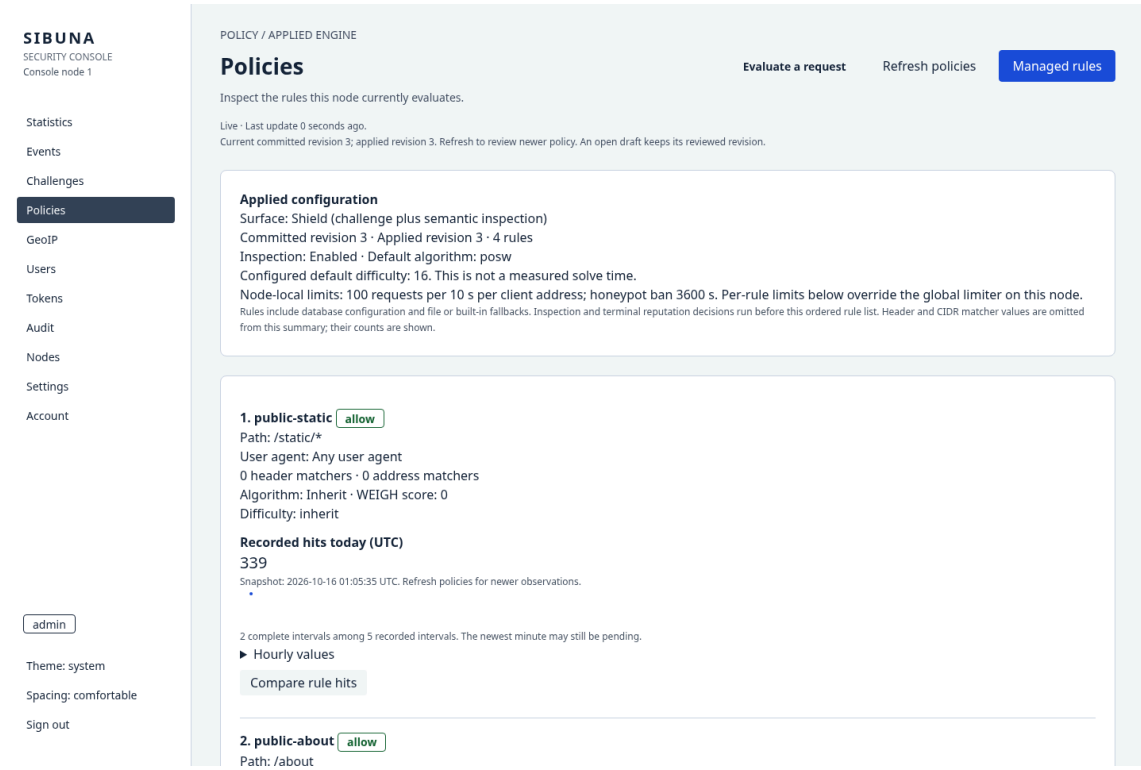


그림 18 정책. 적용된 엔진은 운영 구성, 리비전, 기본 알고리즘과 난이도, 노드 로컬 제한을 밝히고, 규칙을 평가 순서대로 오늘의 적중 수와 그 수의 관측 범위와 함께 나열합니다. 아래쪽에는 검사 모드 표, 요청 테스터, IP 그룹이 있습니다. 관리 규칙을 누르면 편집기가 열립니다. 수정 내용을 저장하면 먼저 바뀐 필드의 검토 화면이 나오고, 확인하면 후보 전체를 검증한 뒤 새 리비전을 만듭니다.

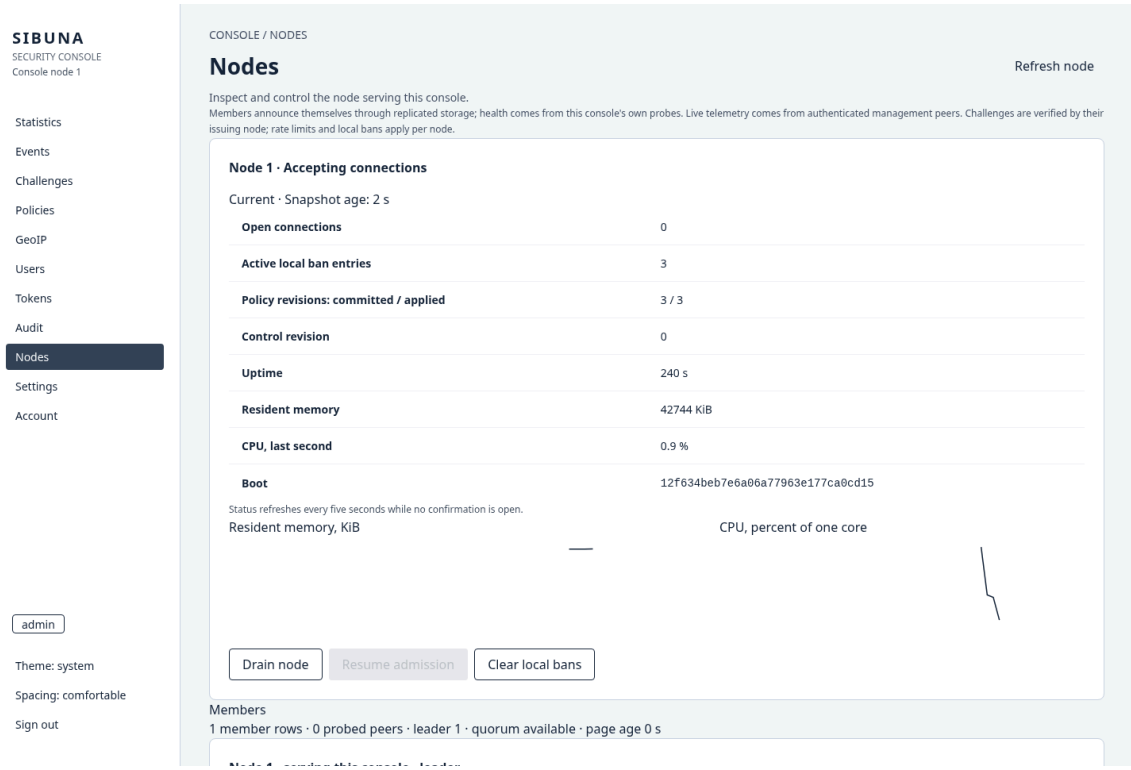


그림 19 노드. 서비스 노드의 상태와 명령, 직전 1초의 상주 메모리와 CPU, 그리고 그 60초 스파크라인을 보여 줍니다. 그 아래에는 자신을 알린 모든 구성원이 적용한 정책 리비전(커밋된 리비전과 비교), 관리 주소, 점검 결과와 함께 나열됩니다(SID 0007, SID 0013).

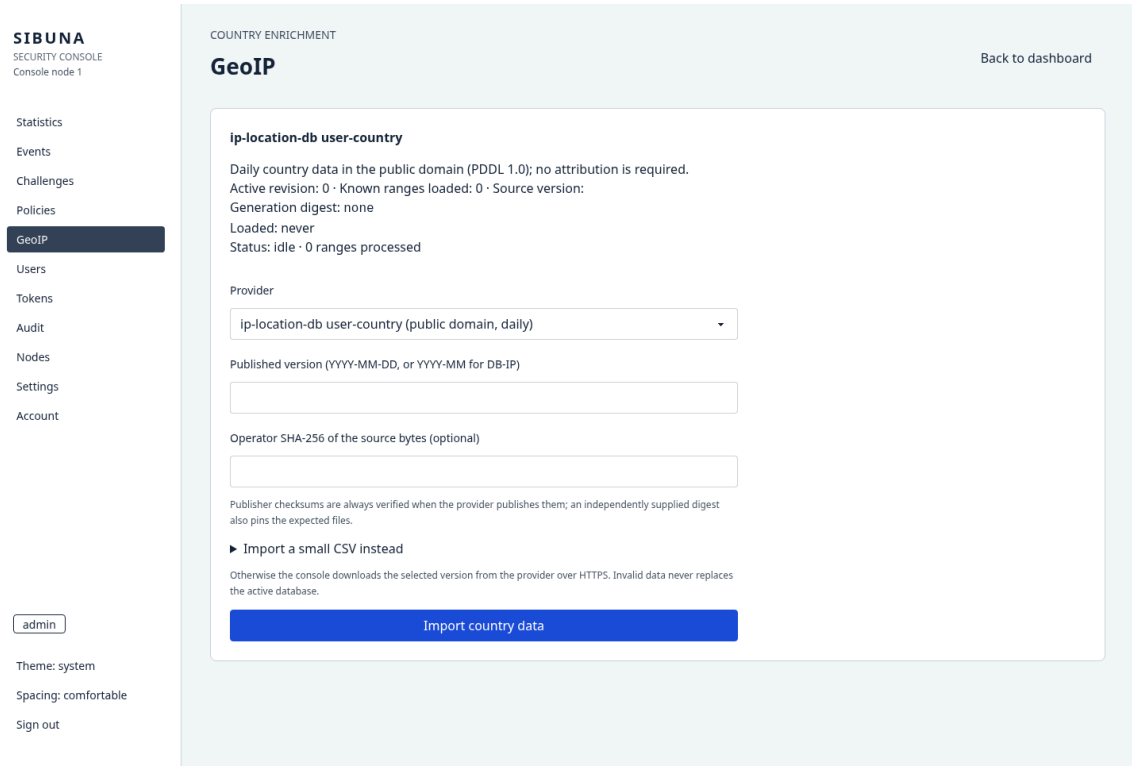


그림 20 GeoIP. 제공자, 라이선스, 활성 세대 다이제스트, 로드 시간, 가져오기 양식을 보여 줍니다. 여기서는 아직 세대를 불러오지 않았습니다. 가져오기에 실패해도 활성 세대는 바뀌지 않습니다.

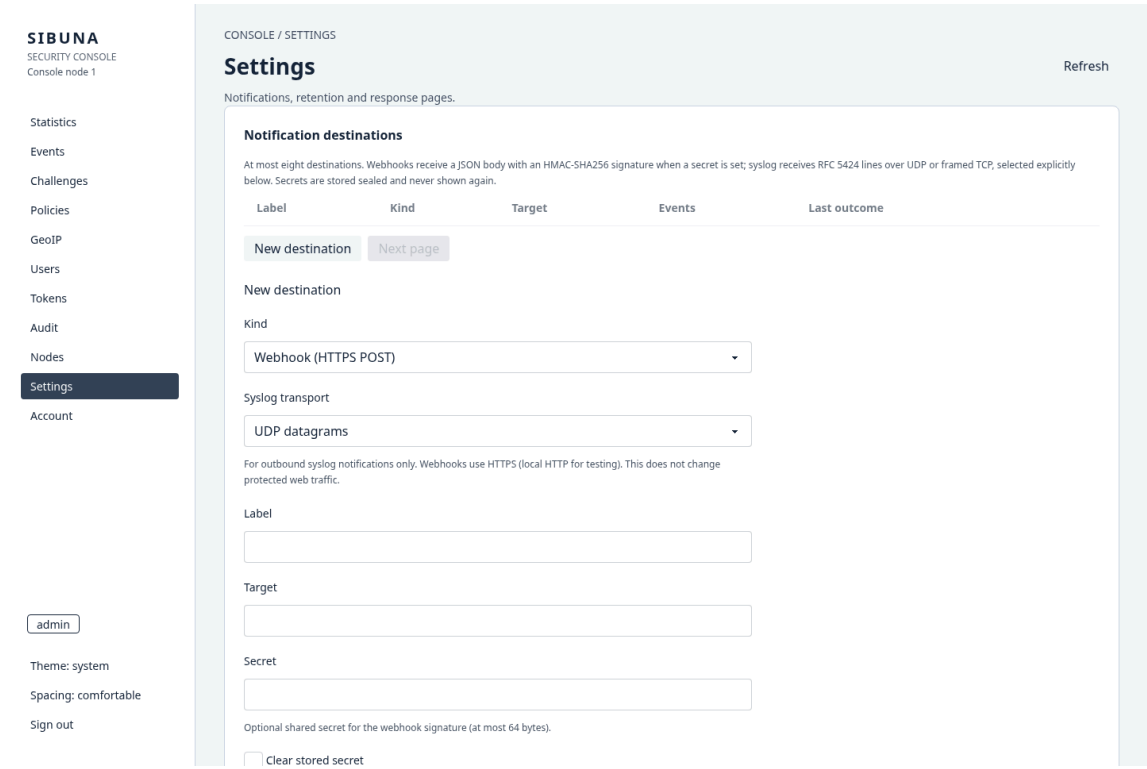


그림 21 설정. 아직 알림 대상이 없고 새 대상을 추가하는 양식이 보입니다. 페이지 아래쪽에는 거부 급증 임계값, 기간별로 명시적 확인을 받는 보존 설정, 응답 페이지 템플릿, 정보 항목이 있습니다.

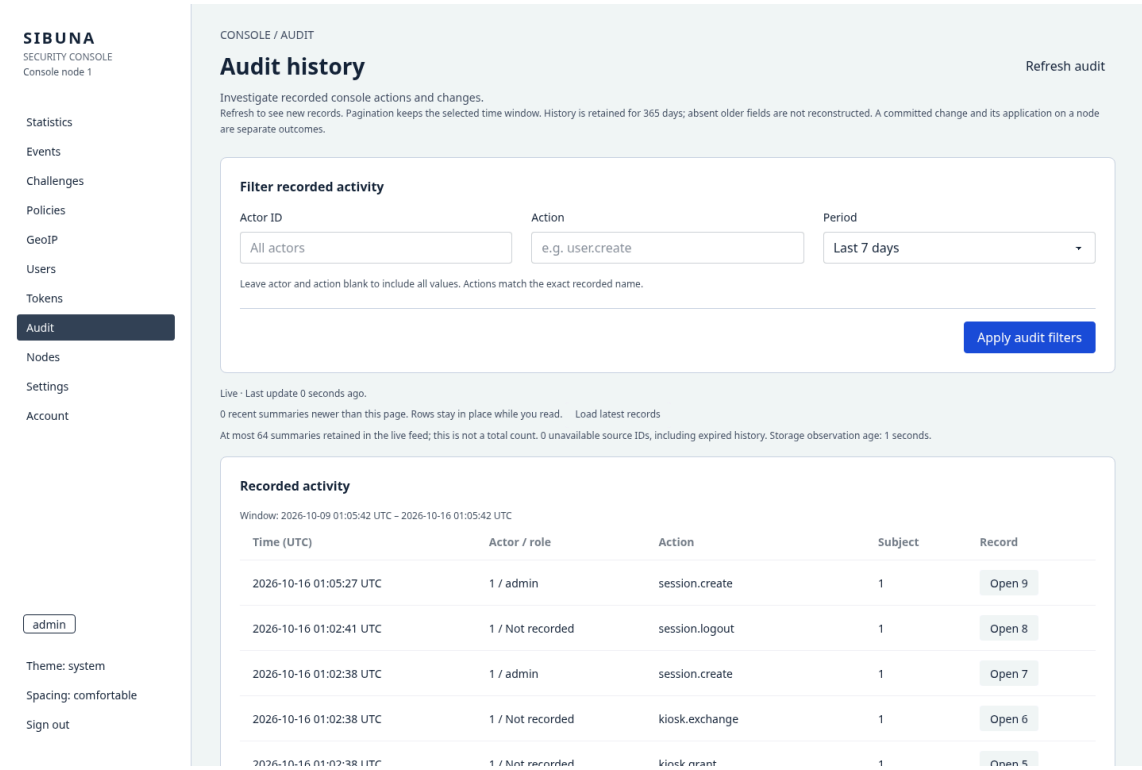


그림 22 감사. 추가만 가능한 기록으로, 수행자와 역할, 동작, 대상, 그리고 마스킹한 변경 전후 요약을 담은 상세 보기를 보여 줍니다. 거부된 로그인도 기록됩니다. 모든 관리 변경과 인증 행은 자격 증명을 제시한 클라이언트 주소를 기록합니다. 시스템이 스스로 쓰는 행에는 주소가 기록되지 않음으로 표시됩니다. 정책 기록에는 이 변경 되돌리기가 있습니다. 확인을 거쳐 해당 리비전 이전에 기록된 문서를 감사되는 새 리비전으로 복원하며, 그 사이 규칙 집합이 바뀌었으면 거부됩니다.

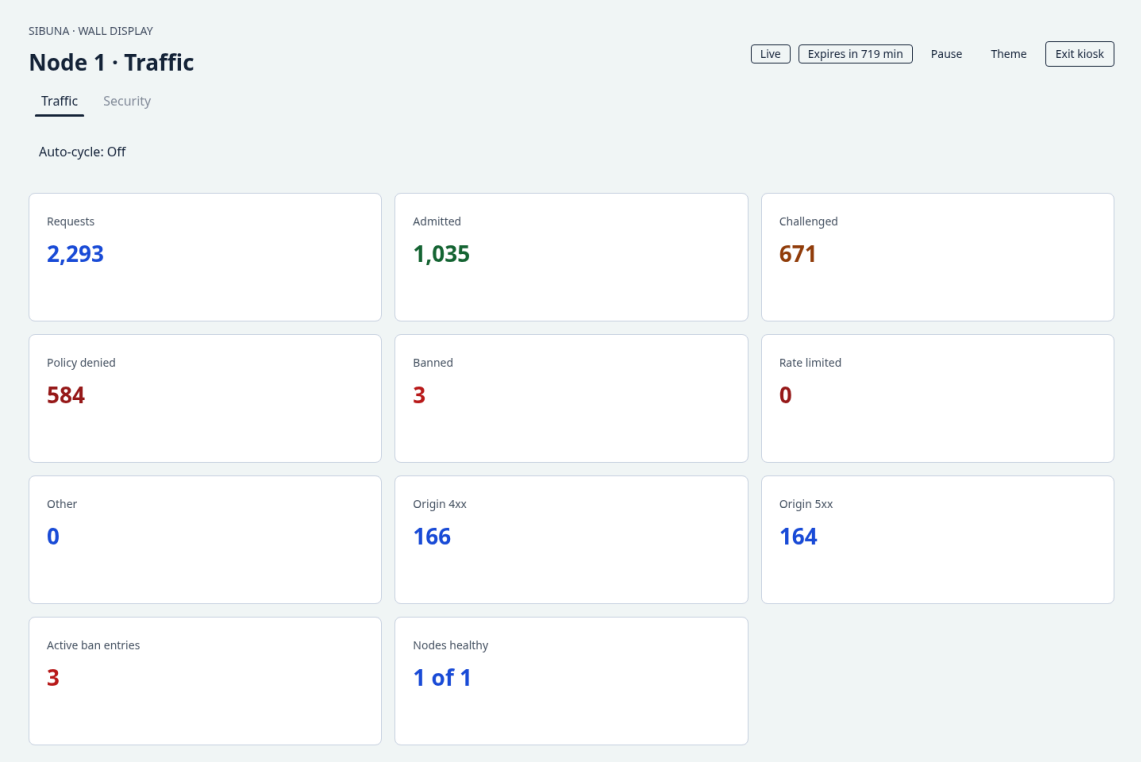


그림 23 벽면 디스플레이. 키오스크 세션은 탐색 메뉴 없이 트래픽과 보안 화면만 보여 주며, 언제 만료되는지 표시합니다.

선택 옵션 `--console-location <latitude,longitude>`는 이 노드의 위치를 선언합니다. 예를 들어 싱가포르의 배포라면 `1.3521,103.8198`입니다. 지구본은 처음에 이 위치를 중심으로 표시되고, Center Sibuna는 회전한 뒤 이 지점으로 돌아갑니다. 국가별 활동은 노드를 향하는 움직이는 대원 호로 표시되며, 뒤쪽 반구와 평면 지도의 날짜 변경선에서 잘립니다. 국가 위치는 대표 중심점이며, 화살표는 개별 연결이 아니라 관측된 60초 구간을 나타냅니다. 서버 위치를 설정하지 않으면 명시적으로 "알 수 없음"으로 남습니다. 설정된 표식은 텔레메트리가 끊긴 동안에도 계속 보이며, 오래된 트래픽은 움직이지 않습니다.

정책 미리보기는 설정된 파일 규칙을 포함한 비공개 후보를 만듭니다. 저장할 때는 예상 리비전을 비교하고 필드 비교 결과를 확인받습니다. 과거 버전으로 되돌리기는 현재 규칙과 비교한 뒤 새 리비전을 만듭니다. 커밋된 리비전과 로컬에 적용된 리비전은 서로 다릅니다. 감사 상세 정보는 크기가 제한된 결정 변경 내역과 실제로 동작한 역할을 보관합니다. 매처 값은 마스킹되며, 문맥이 빠진 오래된 기록은 정보가 없다고 명시됩니다. 드레인, 재개, 로컬 차단 해제에는 명령 미리보기가 필요합니다. 노드에서 다른 명령이 실행 중일 때 도착한 명령은 큐에 쌓이지 않고 `503 CONSOLENODE`로 거부됩니다. 응답을 잃었다면 다시 시도하기 전에 영속 영수증을 확인하세요. 커밋된 의도만으로는 실행 시점의 효과가 끝났다고 할 수 없습니다.

원격으로 운영하려면 HTTPS 프록시를 사용하고 `--console-origin`, `--console-behind-proxy`, 명시적인 `--console-trusted-proxy` CIDR, `--console-key-file`을 설정하세요. 키 파일은 16진수 문자 64개를 담고, 소유자 전용 권한을 가지며, 저장된 2단계 인증 비밀 값을 보호합니다. 재시작해도 유지하되 챌린지 시드와는 따로 보관하세요. 루프백 밖에서 접근하려면 HTTPS 출처와 프록시 허용

목록을 반드시 설정해야 합니다. Sibuna는 콘솔 리스너에서도 들어오는 TLS를 종료하지 않습니다. 콘솔은 나가는 연결에만 rustls를 통해 TLS를 사용합니다(SID 0012).
네이티브 CLI는 같은 권한 부여 및 저장소 규약을 거쳐 실행 중인 콘솔에 접근합니다. 자격 증명은 명령줄 값이 아니라 비공개 파일에서 읽습니다:

```
sibuna console geoip update --version 2026-09-09 \
    --origin http://127.0.0.1:19446 --username admin \
    --password-file ./admin-password
```

소유자 전용 비밀번호 파일을 사용하고, 먼저 비밀번호 설정을 마친 뒤, 필요하면 인증 앱 코드나 복구 코드를 위한 `--factor-file`을 추가하세요. `--token-file`은 계정과 GeoIP 작업에서 사용자 이름과 비밀번호를 대신합니다. 클라이언트는 HTTP/1.1을 사용하고, 리터럴 루프백 주소가 아니면 HTTPS를 요구하며, 리디렉션을 따르지 않고, 요청마다 20초, 응답마다 16 KiB로 제한합니다. `geoip status`는 활성 세대를 읽습니다. 업데이트는 선택한 제공자가 게시한 파일을 HTTPS로 내려받고(기본값은 퍼블릭 도메인 `user-country` 데이터셋, `--provider dbip --version YYYY-MM`을 지정하면 DB-IP Lite), 게시자의 체크섬(또는 `--checksum`으로 지정한 체크섬)을 검증하고, `sibuna-geoip`로 크기가 제한된 범위를 검증한 뒤, 새 세대를 영속 저장하고 로컬에서 활성화합니다. 업데이트에 실패하면 이전 세대가 유지됩니다. CLI 타임아웃(`--timeout`, 기본값 1200초)은 기다리기를 멈출 뿐, 제출된 저장소 작업을 취소하지는 않습니다. DB-IP Lite는 CC BY 4.0 저작자 표시가 필요하며, 이 표시는 해당 데이터가 활성일 때만 나타납니다. `SIBUNA_GEOIP_DATA=<snapshot>`으로 빌드하면 검증된 스냅숏(`geoip-snapshot` 도구가 작성)을 첫 영속 가져오기 전까지 내장합니다. 콘솔이 없는 빌드는 이를 `GEOIP001`로 거부합니다. 알 수 없는 주소, 표본 손실, 오래된 데이터는 그대로 표시됩니다. 국가 데이터를 가져와도 트래픽이 생기거나 이미 만료된 표본이 보강되지는 않습니다.
클러스터 구성원은 시작할 때, 매분, 그리고 적용된 정책 또는 제어 리비전이 바뀐 뒤 2초 안에 공유 데이터베이스의 구성원 테이블(`console_nodes`)에 자신을 알리며, 모든 콘솔의 노드 페이지에 나타납니다. 리더 선출은 없습니다. SQLite 노드는 역할 `single`과 자기 자신을 리더로 보고하고, PostgreSQL 노드는 역할 `unknown`과 리더 없음을 보고합니다. PostgreSQL의 장애 조치는 Sibuna에 보이지 않기 때문입니다(SID 0007). 이 콘솔이 데이터 평면 리스너의 상태를 점검할 피어는 `--console-probe <node-id>=<http://ip:port>`로, 다른 콘솔이 보여 줄 링크는 `--console-advertise <origin>`으로 설정하세요. 피어에서 직접 받는 실시간 텔레메트리는 `--console-peer <node-id>=<https://origin>` (반복 지정 가능, 최대 8개)와 전용 `--console-peer-key-file`을 사용합니다. 피어에는 `--console-behind-proxy`가 필요합니다. 양쪽 엔드포인트가 서로를 나열하고 신뢰할 수 있는 HTTPS 인그레스를 사용해야 합니다. 피어 키는 16진수 문자 64개를 담은 소유자 전용 파일이며, 콘솔 키 및 챌린지 시드와 독립적으로 준비합니다. 피어는 인증된 TLS WebSocket으로 통신하며, `ring` 제공자를 쓰는 rustls를 사용합니다(SID 0012). `--console-peer-ca-file`은 비공개 관리 PKI를 위한 PEM 신뢰 앵커를 선택적으로 제공합니다. 지정하지 않으면 클라이언트는 시스템 루트 인증서를 사용합니다. 호스트 이름 검증은 항상 적용되며 끌 수 없습니다. 각 관리 출처의 호스트(DNS 이름 또는 IP 주소 리터럴)는 피어 인증서에 명시되어 있어야 합니다. 인증서가 거부되면 피어를 `rejected`로 표시하고 `peer.rejected` 감사 행 하나를 씁니다. 키를 교체하거나 CA 파일을 바꾸려면 조율된 재시작이 필요합니다. 피어 인증서는 재시작 없이 갱신할 수 있습니다. 노드 API와 그 구독은 수신 경과 시간, 시계 오차, 부팅 변경, 표본 손실을 보고합니다. 누락된 관측값은 사용할 수 없음으로 남고, 연결이 끊긴 값은 오래된 값으로 남습니다. 받은 통계가 다른 노드 자신의 기여로 바뀌는 일은 없습니다. 대시보드의 실시간 트래픽 범위는 노드 하나를 선택

하거나 설정된 노드를 합칩니다. 노드 범위와 위치는 누락되거나, 오래되었거나, 시계가 어긋난 소스와 관측 시각, 설정된 대상을 식별합니다. 국가 순위는 빠진 소스 행에서 오는 불확실성을 보여 줍니다. 화살표는 수신 노드를 유지하며, 누락된 노드 위치를 지어내지 않습니다. 기여하는 소스 중 하나라도 연속 구간이 없으면 합산 속도는 관측되지 않음으로 남습니다. 노드 하나를 선택하면 그 노드의 보존된 초 단위 기록과 현재 분의 경로 순위를 살펴볼 수 있습니다. 원격 질의는 인증된 피어 연결을 사용합니다. 질의 응답은 4 KiB WebSocket 메시지 하나에 들어가야 하므로, 서비스하는 콘솔은 들어가지 않는 페이지의 행 수를 절반으로 줄입니다. 누락된 피어는 사용할 수 없음으로 남고, 기록 커서는 노드 재시작을 넘어갈 수 없습니다. 분 단위 기록은 자체 노드 선택과 노드별 행을 유지합니다. 탐색 메뉴와 모바일 헤더는 선택한 트래픽 소스와 관계없이 서비스하는 콘솔 노드를 표시합니다. 로컬 명령은 계속 노드별로 동작합니다.

트래픽 타일은 선택한 노드의 보존된 마감 분 단위 기록 가운데 최근 24시간을 먼저 보여 줍니다. 기간 선택에서는 1시간, 7일, 90일, 실시간 부팅 이후 합계도 고를 수 있습니다. 어제 대비 편차를 계산하려면 선택한 모든 노드에서 일치하는 범위가 완전해야 합니다. 지구본과 스파크라인은 여전히 별도의 실시간 60초 구간을 나타냅니다. 보존 기록 스캔은 완료 1분 뒤에 갱신되며, 교체될 때까지 이전 값과 경과 시간을 계속 보여 줍니다. 소스나 기간을 바꾸면 이전 범위는 버립니다. 누락된 기록이 0이 되는 일은 없습니다.

보존된 트래픽 비교는 마감된 분 단위 구간 두 개를 나란히 보여 줍니다. 어제의 같은 구간, 직전 기간, 또는 다른 노드와 비교할 수 있습니다. 길이와 종료 오프셋 설정으로 두 UTC 범위를 고정합니다. 동작 한 번은 한쪽마다 압축 페이지를 최대 16개까지 읽으며, 페이지마다 저장된 기록이 96개 있습니다. 이 정도면 재시작이 겹치지 않는 24시간 구간을 한 번에 다룰 수 있습니다. 더 긴 스캔에서는 계속하기가 같은 경계를 유지합니다. 완전한 범위, 불완전한 범위, 누락된 기록은 명시적으로 표시됩니다. 비율 백분율은 같은 UTC 구간에서 겹치지 않는 완전한 구간이 필요하며, 관측된 밀리초로 정규화합니다. 기준값이 0인데 새 트래픽이 있으면 "새로 생김"으로 표시합니다. 기본 트래픽 기간은 독립적으로 유지됩니다. 분 단위 형식은 과거의 프록시 모드를 기록하지 않으므로, 원본 서버 응답 비교는 forward-auth의 0을 관측된 응답으로 해석하지 않고 사용할 수 없음으로 남습니다.

보존된 경로 순위 비교는 마감된 구간 두 개 또는 노드 두 개를 선택합니다. 노드 0은 운영에서 제외된 구성원을 포함해 보존된 모든 노드를 포함합니다. 동작 한 번은 변경 불가능한 아카이브를 최대 16개까지 읽고, 식별자와 체크섬을 확인하고, 표시할 행을 고르기 전에 모든 카운터를 병합합니다. 아카이브는 늦게 도착하는 표본을 기다리는 60초 구간이 지난 뒤 봉인되므로, 가장 최근에 마감된 분은 아직 대기 중일 수 있습니다. 계속하기는 원래 구간과 커서를 유지합니다. 경계값은 정확한 요청 합계가 아니라 표본으로 수집한 경로 접두사를 나타냅니다. 페이지는 부분 스캔, 잘림, 보존 경계, 보고된 큐 손실을 보여 줍니다. 누락된 아카이브로 트래픽이 0이었다거나 범위가 완전하다고 결론 낼 수는 없습니다.

보안 화면은 최근 24시간을 먼저 보여 주며, 선택한 기간을 같은 길이의 시간 구간 60개로 나눕니다. 집계는 작업량 상한 안에서 저장소 스레드가 실행합니다. SQLite에서 상한은 가상 머신 단계 `--console-query-steps`개입니다(100,000에서 50,000,000까지, 기본값 400만). `--console`과 함께 설정하세요. 잘못되었거나 반복된 값은 거부됩니다. PostgreSQL에서 상한은 읽기 트랜잭션 안에서 설정하는 2,000 ms의 문장 타임아웃입니다(SID 0013). `--console-query-steps`를 바꿔도 이 값은 바뀌지 않습니다. 상한을 다 쓴 질의는 더 좁은 기간을 고르라고 안내합니다. 단계 수는 질의 작업량을 제한할 뿐 경과 시간을 제한하지는 않습니다. 차트는 같은 눈금을 공유합니다. 추세 값을 펼치면 UTC 구간 시작 시각과 정확한 그룹별 수를 볼 수 있습니다. 이 값은 보존된 탐지 결과이므로, 인시던트 범위가 빠져 있다고 해서 공격이 없었다고 해석할 수는 없습니다.

벽면 디스플레이는 키오스크 세션을 사용합니다. 운영자는 계정 화면에서 디스플레이 이름을 정하고 디스플레이 코드 만들기를 선택합니다. 디스플레이는 10분 안에 일회용 코드를 로그인 양식에 붙여 넣습니다. 그 세션은 읽기 전용이고, 통계로만 제한되며, 12시간 안에 만료됩니다. 계정 페이지는 다른 화면으로 이동하거나 페이지가 숨겨지면 표시된 코드를 지웁니다. 숨겨도 사용하지 않은 권한이 취소되지는 않으며, 교환 기한은 그대로 적용됩니다. 코드는 URL에 넣으면 안 됩니다. 디스플레이는 트래픽과 보안 화면을 보여 줍니다. 보안 화면은 인시던트 주소나 페이로드 증거 없이 모듈별 집계 추세와 요청 결과만 보여 줍니다. 자동 순환은 선택 사항이며, 처음에는 꺼져 있고, 동작 줄이기 설정, 오래된 데이터, 일시 정지 중에는 멈춥니다.

알림 대상(거부 급증, 차단, 연결할 수 없는 구성원, 리더 변경에 대한 서명된 웹훅과 syslog)은 설정 아래에 있습니다. 웹훅 비밀 값에는 `--console-key-file`이 필요합니다. 리더 변경 종류는 저장된 대상의 의미가 유지되도록 정의된 채로 남아 있지만, 리더 선출이 없으므로 어느 노드도 이 알림을 내지 않습니다. 대상마다 자체 쿨다운과 3회 재시도 예산이 있습니다. 전달 감사 기록은 결과를 보여 줍니다. 완료된 큐 기록은 7일과 이벤트 4,096개로 제한됩니다. 웹훅에는 안정적인 `Idempotency-Key`가 포함되어 있어, 네트워크 완료 여부가 불확실할 때 수신자가 반복된 효과를 억제할 수 있습니다. 웹훅은 HTTPS로 보내며, 평문 HTTP는 리터럴 루프백 주소로만 보냅니다. 응답 본문은 64 KiB까지 읽습니다. syslog의 UDP/TCP 선택은 나가는 알림에만 적용되며, 보호되는 웹 트래픽과는 관계없습니다. 수동 테스트는 의도와 완료를 감사 기록에 남기고 대상 결과를 갱신합니다. 감사 완료가 확인되지 않으면 운영자가 재시도하기 전에 그 사실을 명시적으로 보여 줍니다.

설정에서는 보존 기간도 제어합니다. 분 단위 기록은 1일에서 90일, 순위는 1일에서 7일, 인시던트는 1일에서 30일, 감사 기록은 1일에서 365일입니다. 기본값은 각 최댓값이며, 순위는 512 MiB 할당량을 유지합니다. 정리 작업이 오래된 기록을 제한된 묶음 단위로 영구 삭제하므로, 보존 값을 저장하려면 확인이 필요합니다. 나중에 값을 늘려도 삭제된 기록은 복원되지 않습니다. 오래된 편집은 거부되며, 성공한 변경은 감사 기록에 나타납니다.

응답 페이지(챌린지, 거부, 속도 제한, 차단, 과부하)는 설정에서 고정된 자리 표시자를 쓰는 크기 제한 HTML로 편집할 수 있습니다. 초안은 샌드박스 탭에서 미리 보며, 저장된 페이지는 다음 정책 스냅숏부터 제공됩니다.

정책 페이지의 정책 작업: 관리 규칙의 순서를 바꾸고, 보존된 검사 탐지 결과에 초안을 재생해 보고, 활성 GeoIP 세대에 고정된 IP 그룹과 국가 차단을 관리하고, 관리 규칙 집합을 내보내거나 원자적으로 가져옵니다(`sibuna console policies export`와 `sibuna console policies import --file <set.json>`도 사용 가능). 가져오기는 요청 하나로, 각각 최대 4 KiB이고 합계 최대 256 KiB인 문서 1개에서 128개를 담습니다. 약 96 KiB를 넘는 집합에 대해서는 페이지의 안내문이 명령줄 가져오기를 알려 줍니다. 적용된 규칙에는 오늘 기록된 적중 수, 시간별 스파크라인, 접근성을 갖춘 표가 표시됩니다. 적중이란 선언형 매처 평가가 성공한 것을 말합니다. 일치한 WEIGH 규칙과 첫 번째 종결 일치가 집계됩니다. 그보다 앞선 검사 또는 평판 결정이 평가를 막을 수도 있습니다. 비공개 테스트는 이 카운터를 늘리지 않습니다. 규칙 적중 비교는 기록된 노드 하나에 대해 마감된 기간 두 개를 고정합니다. 리비전 기록에는 이 편집 전후 적중 비교가 있으며, 편집이 일어난 분을 제외하고 선택한 길이까지 같은 길이의 사용 가능한 기간을 사용합니다. 선택적인 적용 리비전 필터는 관련 없는 세대를 비교에서 제외합니다. 시작, 전환, 누락된 쓰기, 보존 정리는 범위에 눈에 보이는 공백을 남깁니다. 백분율은 양쪽 모두 범위가 완전해야 합니다. 규칙 관측과 시간별/일별 요약은 분 단위 보존 설정을 따릅니다. 편집 전후에 관측된 변화는 비교일 뿐, 그 편집이 트래픽 변화를 일으켰다는 증거가 아닙니다.

더 새로운 GeoIP 세대를 가져와도 국가에서 도출한 기존 평판 행이 자동으로 갱신되지는 않습니다. 국가 동작을 미리 보며 추가, 유지, 제거되는 접두사를 비교하세요. 적용하기 전에 검토한 차이를 페이지 단위로 넘겨 보세요. 교체는 해당 국가가 소유한 더 이상 유효하지 않은 행을 제거하고, 독립적으로 관리되는 접두사는 유지합니다. 세대나 정책 리비전이 바뀌면 미리보기를 새로 해야 하며, 겹치는 독립 편집은 거부됩니다. 이벤트 페이지에는 보존된 WAF 탐지 결과와 허니팟 인시던트가 있습니다. 완전한 접근 로그는 아닙니다. GeoIP를 불러온 상태라면 저장소 스레드가 인시던트 묶음의 트랜잭션 안에서 각 인시던트의 국가와 세대를 기록합니다. 이는 영속 저장 시점의 귀속이지, 요청 시점의 위치를 재구성한 것이 아닙니다. 이후의 가져오기는 기록된 매핑을 바꾸지 않습니다. 국가 필터는 대문자 두 글자 코드, 불러온 세대에 없는 주소를 뜻하는 unknown, 매핑 데이터가 없는 인시던트를 뜻하는 not_recorded를 받습니다. 소스 그룹은 국가가 섞여 있거나 범위가 불완전하면 그렇다고 명시적으로 보고합니다. 지구본의 이벤트 보기 동작은 선택한 노드를 유지한 채 해당 국가의 최근 1시간 보존 인시던트를 엽니다. 통계 페이지의 보안 화면은 실시간 속도 제한, 챌린지, 차단 비율을 1시간, 1일, 7일, 30일 동안의 보존된 검사 및 허니팟 탐지 결과와 함께 보여 줍니다. 범주, 소스, 경로 링크는 같은 노드와 고정된 시간 경계로 일반 인시던트 작업 화면을 엽니다. 인시던트 필터를 적용하면 새 기간이 시작됩니다. 탐지 결과에는 감사 기록이 포함되며, 차단된 요청 합계와는 다릅니다. 평판과 규칙 적중 귀속이 없으면 기록되지 않음으로 표시됩니다. 이벤트와 감사 필터는 라벨을 해당 컨트롤과 함께 두고, 적용 동작을 따로 정렬하며, 사용 가능한 콘텐츠 너비에 맞춰 열을 조정합니다. 인증된 WebSocket 하나가 화면 이동 후에도 유지되며 통계, 인시던트 요약, 노드 상태, 정책 리비전, 챌린지, 감사 요약을 전달합니다. 새 인시던트와 감사 기록은 최신 기록 불러오기 뒤에서 기다리므로, 표를 읽는 동안 표가 제자리에 머뭅니다. 정책 업데이트는 열려 있는 초안을 대체하지 않고 현재 커밋된 리비전과 적용된 리비전을 보여 줍니다. 흐름 카운터는 실시간으로 갱신됩니다. 기본값이 아닌 챌린지 시간 구분을 선택했다면 명시적으로 새로 고치세요. 과거 기록 질의, 상세 읽기, 변경 작업은 계속 HTTP 요청으로 처리합니다.

페이지 조각은 북마크할 수 있으며, 뒤로 가기와 앞으로 가기는 인증된 페이지를 다시 엽니다. 사이드바는 이 브라우저에서 테마와 간격 선택도 기억하며, 기본 테마는 시스템입니다. SID 0007은 Committed 상태입니다. 콘솔은 골든 픽스처와 헤드리스 Chromium의 브라우저 작업 흐름 테스트를 통과합니다(cargo xtask browser console, SID 0010 기준 5). 종단 간 테스트 모음은 cmd/sibuna/tests/console_*.rs이며, 그 PostgreSQL 클러스터 시나리오는 SIBUNA_TEST_DATABASE_URL을 설정했을 때 실행됩니다. SID 0010은 콘솔을 컴파일해 넣고 비활성화했을 때, 저장소만 켰을 때, 활성화했지만 유휴 상태일 때 모두 16개 제품 워크로드가 콘솔 없이 빌드한 데몬의 약 1% 이내, 즉 잡음 범위 안에 머문다고 기록합니다. 부하 상태에서 대시보드 8개를 활성화하는 SID 0007의 공식 격리 기준은 아직 측정하지 않았습니다(제VIII부). 전체 WebAssembly 애플리케이션에는 압축하지 않은 크기 기준 768 KiB 상한이 있으며, 데몬의 빌드 스크립트가 빌드를 실패시켜 이를 강제합니다. 검토 임계값은 640 KiB입니다. 1.0.0-beta.3에서 모듈 크기는 724,330바이트(gzip 압축 272,568바이트)로 검토 임계값을 넘으며, SID 0007은 이를 검토 대상으로 기록합니다. 이 프로젝트 제한은 콘솔 작업 흐름을 위한 여유를 남겨 둘 뿐, 브라우저 로딩과 응답성 측정을 대신하지는 않습니다. 모듈의 선형 메모리는 필요할 때 선언된 최대 16 MiB까지 늘어납니다.

배포 토폴로지

학습 목표 독립형 리버스 프록시와 forward-auth 검증기를 Nginx 또는 Caddy 뒤에 배포합니다.

리버스 프록시

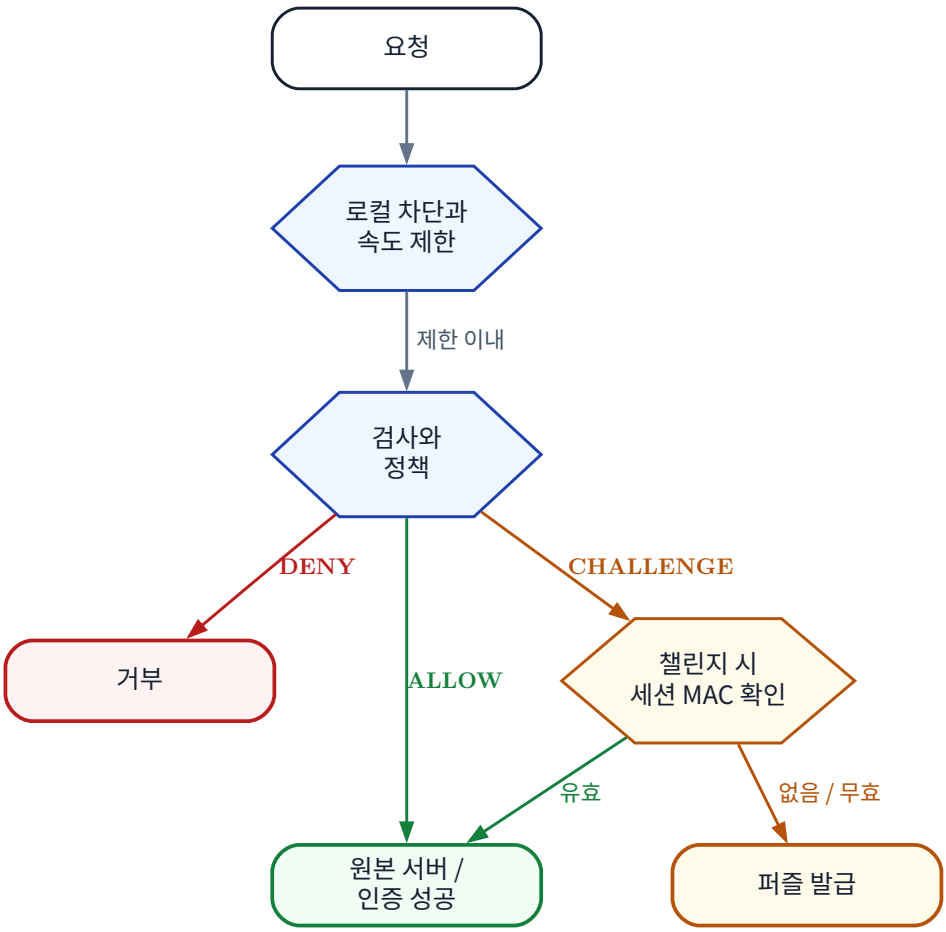


그림 24 Shield 운영 구성의 요청 라우팅

```
sibuna --port 80 --upstream-host 127.0.0.1 --upstream-port 3000 \
       --secret-file /etc/sibuna/secret --difficulty 16 --algorithm posw
```

허용된 요청은 `X-Forwarded-For`, `X-Forwarded-Proto`, `X-Real-IP`, `X-Sibuna-Status`, `X-Sibuna-Rule` 헤더와 함께 원본 서버에 도달합니다. 홉별 헤더, Sibuna가 직접 쓰는 필드, 대체 전달 필드(`Forwarded`, `X-Forwarded-Host`, `X-Forwarded-Port`)는 제거되며, 클라이언트가 이름을 밑줄로 쓴 경우(`X_Forwarded_For`)에도 제거됩니다. `X-Real-IP`는 요청을 판정할 때 사용한 클라이언트입니다. `X-Forwarded-For`는 소켓 피어만 담으며, 클라이언트가 쓴 항목은 절대 담지 않습니다. 전달된 신원을 신뢰하는 경우

에는 받은 체인 뒤에 소켓 피어를 붙인 값입니다. 따라서 원본 서버는 클라이언트를 `X-Real-IP`나 `X-Forwarded-For`의 오른쪽에서 읽어야 하며, 가장 왼쪽 항목에서 읽으면 안 됩니다. 원본 서버가 받는 경로는 정책이 판정한 정규형입니다(SID 0009). 정규형이 없는 경로에는 `400`으로 응답합니다.

HTTP/1.1 WebSocket 업그레이드는 원본 서버의 핸드셰이크를 받아들이기 전에 요청 허용과 정책 검사를 거칩니다. 하위 프로토콜과 확장 협상은 종단 간에 그대로 유지됩니다. 스레드 두 개가 방향마다 하나씩, 각각 고정 버퍼 하나로 미리 읽은 바이트와 반쪽 닫기까지 포함해 바이트를 중계합니다. 따라서 업그레이드된 연결은 스레드 두 개를 차지하며, 모든 연결이 업그레이드되면 연결 스레드 수가 `--max-connections`의 두 배까지 늘 수 있습니다. 업그레이드된 소켓은 일반 연결 할당량과 종료 레지스트리 안에 남지만, HTTP 유휴 기한과 별개로 `--websocket-idle-timeout`(기본값 300초)을 사용합니다. 어느 방향의 트래픽이든 이 기한을 갱신하지만, 기간 내내 한 바이트도 받지 않는 피어는 반대 방향이 아무리 바빠도 중계를 끝냅니다. 핸드셰이크 이후의 프레임은 WAF 검사 입력이 아닙니다. HTTPS/WSS는 Sibuna의 비공개 HTTP/1.1 리스너 앞에서 TLS를 종료하는 인그레스를 사용합니다. Sibuna는 브라우저 TLS를 직접 종료하지 않습니다. 인그레스는 브라우저와 HTTP/2를 협상하고 Sibuna에는 HTTP/1.1로 전달할 수 있습니다. Sibuna의 네이티브 HTTP/2 지원은 이후 업데이트로 미뤄졌습니다. 이 구성은 네이티브 gRPC 같은 애플리케이션에 종단 간 HTTP/2 의미론을 제공하지 않습니다.

Content-Length 요청 본문은 바이트와 MIME 헤더를 보존하며 고정 버퍼를 거쳐 스트리밍됩니다. 멀티파트 업로드는 경계, 반복된 필드 이름, 파일 이름, 파트 유형을 유지합니다. WAF는 본문의 처음 8 KiB까지만 검사합니다. 멀티파트 메타데이터와 파일이 아닌 필드는 텍스트 검사 입력으로 남고, 파일 페이로드는 불투명하게 취급됩니다. `filename`이 비어 있는 파트는 일반 필드입니다. 인식된 바이너리 최상위 MIME 유형(SVG를 제외한 이미지, 오디오, 비디오, PDF, ZIP, gzip, 7z, protobuf, octet-stream)의 본문은 접두사도 바이너리처럼 보일 때만 불투명하게 취급합니다. 즉 16바이트 중 적어도 1바이트가 탭, 줄 바꿈, 수직 탭, 폼 피드, 캐리지 리턴이 아닌 제어 바이트여야 합니다. 그렇지 않으면 텍스트로 검사합니다(SID 0004). JSON, XML, SVG, URL 인코딩 양식, 알 수 없는 유형은 텍스트 검사를 유지합니다. 모호하거나 형식이 잘못된 MIME 메타데이터는 텍스트 검사로 돌아갑니다. 멀티파트 파싱은 같은 접두사 안에서 파트 32개와 파트당 헤더 2 KiB로 제한되며, 구분자 뒤의 RFC 2046 전송 패딩을 허용합니다.

이는 업로드 호환성일 뿐, 파일 검증이나 악성 코드 검사가 아닙니다. 백엔드는 클라이언트의 Content-Type 선언을 믿지 말고 허용하는 미디어 유형을 직접 강제해야 합니다. 검사 접두사 뒤의 필드는 큰 업로드 파일 뒤에 오는 필드를 포함해 검사하지 않습니다. 압축된 페이로드는 검사를 위해 압축을 풀지 않습니다. 업로드도 요청 허용, 경로, 쿼리, 헤더 검사를 거치며 연결 기한의 적용을 받습니다. 청크 요청 본문은 검사 전에 연결 버퍼 안에서 디코딩됩니다(SID 0009). 버퍼 안에서 끝나는 본문은 Content-Length와 함께 백엔드에 도달합니다. 더 긴 본문은 Sibuna가 읽을 때마다 청크 하나로 다시 청크 인코딩하므로, 백엔드는 클라이언트의 청크 크기, 확장, 트레일러를 보지 못합니다. 단독 CR 또는 LF가 있거나, 크기 주변에 공백이 있거나, 확장의 형식이 잘못되었거나, 4 KiB를 넘는 청크 줄은 400으로 거부됩니다. 16 KiB를 넘는 트레일러 섹션도 마찬가지입니다. 다른 코딩을 쓰는 `Transfer-Encoding`은 501을 받습니다. 청크 요청을 파싱할 수 없는 백엔드도 약 44 KiB 미만의 본문은 Content-Length로 받습니다. 요청 본문을 전달한 원본 서버 연결은 그 교환이 끝나면 닫히며 풀로 돌아가지 않습니다. `100-continue`를 기다리는 클라이언트는 본문을 보내기 전에 로컬에서 이를 받습니다. 지원하지 않는 Expect 값은 417을 받습니다. Expect 헤더는 백엔드로 전달하기 전에 소비됩니다.

인그레스 뒤의 forward-auth

--mode forward_auth에서 데몬은 인그레스의 하위 요청에 200 (감사 헤더 X-Sibuna-Status, X-Sibuna-Rule, X-Sibuna-Rule-Hash 포함), 챌린지에는 401, 거부에는 403, 속도 제한에는 429로 응답합니다. 인그레스는 클라이언트 주소와 원래 URL을 전달해야 하며, forward-auth 모드는 기본적으로 이를 신뢰합니다. 인그레스만 연결할 수 있도록 이 리스너를 비공개로 바인딩하세요. X-Forwarded-Uri (Caddy) 또는 X-Original-URI (Nginx)와 X-Forwarded-Method는 정책 평가를 위해 애플리케이션 요청을 복원합니다. 원래 경로는 요청 줄의 경로와 마찬가지로 정규형으로 바뀝니다. 원래 URL 필드가 중복되거나 충돌하는 경우, 그리고 원래 URI가 origin-form이 아닌 경우에는 400으로 거부합니다. 데몬 내부 경로는 항상 실제 요청 URI를 사용합니다. forward-auth는 인그레스가 생략한 본문을 검사하지 않습니다.

```
map $http_upgrade $sibuna_connection_upgrade {
    default upgrade;
    '' close;
}
server {
    listen 443 ssl;
    location / {
        auth_request /__sibuna_auth;
        auth_request_set $sibuna_auth_status $upstream_status;
        auth_request_set $sibuna_retry_after $upstream_http_retry_after;
        auth_request_set $sibuna_status $upstream_http_x_sibuna_status;
        auth_request_set $sibuna_rule $upstream_http_x_sibuna_rule;
        auth_request_set $sibuna_rule_hash $upstream_http_x_sibuna_rule_hash;
        error_page 401 = @sibuna_challenge;
        error_page 500 = @sibuna_auth_error;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $sibuna_connection_upgrade;
        proxy_set_header Host $host;
        proxy_set_header X-Sibuna-Status $sibuna_status;
        proxy_set_header X-Sibuna-Rule $sibuna_rule;
        proxy_set_header X-Sibuna-Rule-Hash $sibuna_rule_hash;
        proxy_read_timeout 300s;
        proxy_pass http://127.0.0.1:3000;
    }
    location = /__sibuna_auth {
        internal;
        proxy_pass http://127.0.0.1:8080/;
        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
        proxy_set_header X-Forwarded-Method $request_method;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Uri "";
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header User-Agent $http_user_agent;
        proxy_set_header Cookie $http_cookie;
    }
    location @sibuna_challenge {
        rewrite ^ /__sibuna/challenge break;
        proxy_pass http://127.0.0.1:8080;
        proxy_pass_request_body off;
```

```
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
        proxy_set_header X-Forwarded-Method $request_method;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
    location @sibuna_auth_error {
        add_header Retry-After $sibuna_retry_after always;
        if ($sibuna_auth_status = 429) { return 429; }
        return 503;
    }
    location /__sibuna/ {
        proxy_pass http://127.0.0.1:8080;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

```
example.com {
    handle /__sibuna/* {
        reverse_proxy localhost:8080
    }
    handle {
        forward_auth localhost:8080 {
            uri /
            header_up X-Forwarded-For {remote_host}
            header_up -X-Original-URI
            copy_headers X-Sibuna-Status X-Sibuna-Rule X-Sibuna-Rule-Hash
        }
        reverse_proxy localhost:3000
    }
}
```

/__sibuna/* 네임스페이스(중간 페이지, 챌린지, 검증, 솔버 리소스)는 두 구성 예 모두에서 데몬에 직접 도달해야 합니다. nginx 오류 페이지는 원래 URI와 메서드를 전달하므로, 중간 페이지는 권한 부여 결정이 정한 요구 수준을 그대로 담습니다. 이 값이 없어도 페이지는 동작하며, 이때 발급은 브라우저가 보고한 URL을 평가하는 방식으로 대체됩니다. Caddy는 원래 URI/메서드 메타데이터를 스스로 제공합니다. 배타적인 handle 블록은 챌린지와 검증 경로가 forward-auth 사전 검사에 들어가지 않도록 합니다. Nginx에서 애플리케이션 WebSocket을 쓰려면 위에 보인 명시적인 Upgrade/Connection 헤더가 필요합니다. Nginx 오류 처리기는 두 번째 권한 부여 결정을 내리거나 업로드 본문을 소비하지 않고 내부 챌린지 경로를 렌더링합니다. Nginx의 auth 모듈은 2xx, 401, 403만 직접 받으므로, 예제는 속도 제한된 auth 오류를 Retry-After와 함께 429로 다시 바꿉니다. 그 밖의 권한 부여 실패는 계속 503으로 막습니다. Nginx는 자체 403 본문을 제공하는 반면, Caddy는 Sibuna의 거부 페이지를 전달합니다. 두 구성 예 모두 들어오는 Sibuna 감사 헤더를 실제 권한 부여 응답으로 교체합니다. 일반 TLS 인증서와 애플리케이션 업로드/기한 제한은 인그레스에서 설정하세요. 예제는 라우팅 논리를 보여 줍니다. 저장소는 아직 이 구성 예를 실제 인그레스에서 테스트하지 않습니다. 구성 예가 의존하는 전달 메타데이터 규칙은 sibuna-http의 단위 테스트가 다룹니다.

인그레스에서 받는 JA4 지문

JA4 지문은 클라이언트가 TLS 연결을 여는 방식을 요약합니다. 이 지문은 클라이언트의 첫 핸드셰이크 메시지인 ClientHello에서 계산합니다. 클라이언트의 TLS 라이브러리는 HTTP 바이트를 하나라도 보내기 전에 이 메시지를 확정하므로, 클라이언트는 HTTP 요청 안에서 자신의 지문을 바꿀 수 없습니다. 브라우저, 라이브러리, 봇은 이 지문이 서로 다릅니다. 그래서 JA4는 어떤 클라이언트든 마음대로 쓸 수 있는 `User-Agent` 값보다 강한 신호입니다(SID 0014). JA4는 FoxIO가 정의했습니다. 지문은 36문자이며, _로 이어진 세 부분으로 이루어집니다.

```
t13d1516h2_8daaf6152771_02713d6af862
^^^^^^^^^^ ^^^^^^^^^^^^ ^^^^^^^^^^^^

a          b            c

a  프로토콜(t는 TCP, q는 QUIC, d는 DTLS), TLS 버전(13은 TLS 1.3),
   SNI가 있으면 d, 없으면 i, 암호 스위트 개수(두 자리),
   확장 개수(두 자리), ALPN 코드(h2는 HTTP/2)
b  정렬한 암호 스위트의 12자리 소문자 16진수 해시
c  정렬한 확장과 서명 알고리즘의
   12자리 소문자 16진수 해시
```

TLS를 종료하는 구성 요소만 ClientHello를 보므로, JA4를 계산할 수 있는 것도 그 구성 요소뿐입니다. Sibuna는 TLS를 종료하지 않습니다. Sibuna는 인그레스 뒤에서 평문 HTTP/1.1로 서비스하고, 브라우저와 TLS, HTTP/2, HTTP/3로 통신하는 것은 인그레스입니다. 여러 인그레스가 이미 JA4를 계산합니다. 그래서 우리는 인그레스가 지문을 계산해 요청 헤더로 보내게 하고, Sibuna는 그 헤더에서 지문을 읽습니다.

`--ja4-header <name>`은 그 헤더의 이름을 지정합니다. 이 옵션은 기본적으로 꺼져 있고 기본 이름도 없습니다. 우리는 항상 인그레스가 쓰는 헤더 이름을 지정합니다. Sibuna는 이름을 소문자로 저장하고 대소문자를 구별하지 않고 비교합니다. Sibuna가 직접 쓰거나 제거하는 필드의 이름(`X-Forwarded-*`, `X-Real-IP`, `X-Sibuna-*`, `Forwarded`, 그리고 - 대신 _를 쓴 형태)은 거부합니다. 옵션을 지정하면 시작 배너에 `JA4: from cf-ja4 (trusted)` 같은 줄이 출력됩니다.

헤더는 전달된 필드를 신뢰할 때만 읽습니다. 이는 `X-Forwarded-For`에 적용되는 것과 같은 결정으로, `--trust-forwarded`를 지정하거나 `--no-trust-forwarded` 없이 forward-auth 모드를 쓰는 경우입니다. 신뢰하지 않는 피어에서 온 헤더는 무시합니다. 신뢰할 때는 그 이름의 필드가 정확히 하나 있고 그 값이 JA4 형식이면 요청에 지문이 있는 것으로 봅니다. 값의 형식이 잘못되었거나 필드가 반복되면 요청에는 지문이 없습니다. 카운터 두 개가 이를 관찰합니다. `sibuna_ja4_present_total`은 지문이 있는 요청을 세고, `sibuna_ja4_invalid_total`은 헤더 형식이 잘못되었거나 반복된 요청을 셉니다. 신뢰하지 않는 피어의 요청은 어느 카운터도 올리지 않습니다.

리버스 프록시 모드에서는 Sibuna가 헤더를 신뢰할 때만 원본 서버가 그 헤더를 봅니다. 신뢰하지 않으면 Sibuna는 원본 서버로 보내는 요청에서 그 이름의 필드를 모두 제거하며, 클라이언트가 이름에서 - 대신 _를 쓴 경우에도 제거합니다. 따라서 클라이언트는 그 이름으로 위조한 지문을 원본 서버에 넘길 수 없습니다. 신뢰할 때는 필드가 정확한 표기 그대로 바뀌지 않고 전달되며, _를 쓴 별칭은 여전히 제거됩니다. `--ja4-header`가 없으면 그 이름의 필드는 평범한 필드입니다.

헤더는 인그레스가 직접 써야 합니다 지문은 그것을 쓴 인그레스만큼만 믿을 수 있습니다. Sibuna는 인그레스가 계산한 값과 클라이언트가 보낸 값을 구별할 수 없습니다. 따라서 인그레스는 Sibuna에 도달하는 모든 경로에서 클라이언트가 보낸 헤더 사본을 덮어쓰거나 제거해야 합니다. `X-Forwarded-For`와 마찬가지로, Sibuna의 리스너는 인그레스의 연결만 받아들여야 합니다.

Cloudflare. 관리형 변환 "Add bot protection headers"는 원본 서버로 보내는 요청에 `cf-bot-score`, `cf-verified-bot`, `cf-ja3-hash`와 함께 `cf-ja4`를 추가합니다. 이 기능에는 Bot Management가 포함된 Enterprise 요금제가 필요합니다. Cloudflare와 Sibuna 사이의 인그레스는 이 필드를 바꾸지 않고 그대로 전달해야 합니다. 다음 명령으로 Sibuna를 시작합니다.

```
sibuna --ja4-header cf-ja4 --trust-forwarded -s /etc/sibuna/secret
```

Sibuna 앞의 인그레스는 Cloudflare의 주소에서 오는 연결만 받아들여야 합니다. 그렇지 않으면 클라이언트가 Cloudflare를 우회해 자신의 `cf-ja4`를 보낼 수 있습니다.

nginx. FoxIO의 `ja4-nginx-module`은 nginx 안에서 JA4를 계산해 `$http_ssl_ja4`로 노출합니다. 이 모듈에는 모듈의 ClientHello 캡처 패치를 적용하고 `--with-http_ssl_module`로 빌드한 nginx 1.26 이상이 필요합니다. `proxy_set_header`는 클라이언트가 보낸 같은 이름의 필드를 대체하므로, Sibuna에 도달하는 각 location이 헤더를 직접 설정합니다.

```
# 리버스 프록시: Sibuna는 --ja4-header x-ja4 --trust-forwarded로 실행
location / {
    proxy_set_header X-JA4 $http_ssl_ja4;
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_pass http://127.0.0.1:8080;
}

# forward-auth: Sibuna는 --mode forward_auth --ja4-header x-ja4로 실행
location = /__sibuna_auth {
    internal;
    proxy_pass http://127.0.0.1:8080/;
    proxy_set_header X-JA4 $http_ssl_ja4;
    # ... 위 forward-auth 구성 예의 나머지 필드
}
```

forward-auth 구성 예에서는 같은 `proxy_set_header X-JA4 $http_ssl_ja4;` 줄이 `@sibuna_challenge`와 `location /__sibuna/`에도 들어가야 합니다. 이 줄을 빠뜨린 location은 클라이언트가 보낸 `X-JA4`를 그대로 Sibuna에 넘기고, Sibuna는 그 값을 신뢰합니다.

Apache Traffic Server. `ja4_fingerprint` 플러그인은 `plugin.config`의 `ja4_fingerprint.so` 줄로 활성화합니다. 이 플러그인에는 옵션이 없습니다. 플러그인은 각 TLS 연결의 요청에 지문을 담은 `ja4` 필드와 프록시 이름을 담은 `x-ja4-via` 필드를 추가합니다. Sibuna는 `--ja4-header ja4 --trust-forwarded`로 시작합니다. 플러그인 문서는 클라이언트가 보낸 `ja4` 필드를 플러그인이 대체하는지 밝히지 않습니다. 값을 신뢰하기 전에, 우리가 쓰는 버전에서 이를 확인하거나 플러그인이 자기 필드를 추가하기 전에 클라이언트의 필드를 제거해야 합니다.

규칙은 `ja4` 필드로 지문을 비교합니다(제X부, 정책 스키마). 값은 정확히 같아야 하는 전체 지문이거나, 지문이 그것으로 시작해야 하는 접두사 뒤에 `*`를 붙인 것입니다. "`*`" 하나만 쓰면 모든

지문과 일치합니다. 지문이 없는 요청은 ja4 규칙과 절대 일치하지 않습니다. 로더는 대문자로 쓴 T13D*처럼 어떤 지문과도 일치할 수 없는 패턴을 INVALID POLICY JA4로 거부합니다. 비교는 바이트 단위로 이루어지며 메모리를 할당하지 않습니다.

```
{ "rules": [
    { "name": "deny-known-scanner", "action": "DENY",
      "ja4": "t13i010400_0f2cb44170f4_5c4c70b73fa0" },
    { "name": "weigh-no-sni", "ja4": "t13i*", "action": "WEIGH", "weight": 15 },
    { "name": "weigh-headless", "user_agent": "Headless", "action": "WEIGH",
      "weight": 30 },
    { "name": "challenge-tls12-checkout", "path": "/checkout/*", "ja4": "t12*",
      "action": "CHALLENGE" }
] }
```

첫 번째 규칙은 지문 하나를 정확히 일치시켜 거부합니다. 이 값은 예시일 뿐 알려진 스캐너가 아닙니다. 다음 두 규칙은 결정하는 대신 요청의 점수에 가중치를 더합니다. 하나는 서버 이름을 보내지 않은 TLS 1.3 클라이언트이고, 다른 하나는 User-Agent로 알아본 헤드리스 브라우저입니다. 뒤의 규칙이 먼저 결정하지 않으면, 두 신호를 모두 보이는 요청은 점수가 45가 되어 기본 거부 임계값 40에 도달하고, 하나만 보이는 요청은 챌린지를 받습니다. 마지막 규칙은 결제 경로에서 TCP에서 TLS 1.2를 협상한 모든 클라이언트에게 접두사로 챌린지를 요구합니다. 정당한 오래된 클라이언트 상당수가 아직 TLS 1.2를 쓰므로 거부하지 않고 챌린지를 요구합니다. 규칙은 이렇게 JA4를 다른 신호와 함께 써야 합니다. JA4 값은 사람이나 기기가 아니라 TLS 라이브러리와 그 설정을 가리킵니다. 많은 클라이언트가 하나의 라이브러리를 공유하며, 특수한 라이브러리는 브라우저의 ClientHello를 흉내 낼 수 있습니다. JA4는 자동화의 비용을 높일 뿐 신원을 증명하지 않습니다. 같은 이유로 Sibuna는 차단과 평판을 지문이 아니라 항상 주소를 키로 관리합니다.

인시던트(WAF 거부, 허니팟 적중, 감사 탐지 결과)는 요청에 지문이 있었으면 그 지문을 security_incidents.ja4 열에 기록합니다. 콘솔은 인시던트 증거에서 이를 "JA4 fingerprint"로 표시하며, 값이나 "Not recorded"를 보여 줍니다. 이벤트 필터는 지문이나 접두사를 받습니다. 관리 규칙 편집기에는 policies.ja4_pattern에 저장되는 JA4 필드가 있고, 정책 테스트 양식은 지문을 입력받습니다.

선언형 정책

> **학습 목표** 규칙, 가중치, 임계값, IP 규칙, WAF 스위치를 담은 정책 파일을 작성합니다.

```
{
  "default_action": "CHALLENGE",
  "waf": true,
  "thresholds": { "challenge_at": 10, "deny_at": 40, "bits_step": 5 },
  "ip_rules": { "10.0.0.0/8": "ALLOW", "192.0.2.0/24": "DENY", "2001:db8::/32": "DENY" },
  "rules": [
    { "name": "deny-cf-workers", "headers": { "CF-Worker": ".*" }, "action": "DENY" },
    { "name": "deny-amazonbot", "user_agent": "Amazonbot", "action": "DENY" },
    { "name": "api-with-key", "path": "/api/*", "headers": { "X-Api-Key": ".*" },
    "action": "ALLOW" },
```

```
    { "name": "protect-checkout", "path": "/checkout/*", "action": "CHALLENGE",
      "challenge": { "difficulty": 20, "algorithm": "posw" } },
    { "name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight": 30 },
    { "name": "internal-vpc", "remote_addresses": ["10.0.0.0/8", "fd00::/8"],
    "action": "ALLOW" }
  ]
}
```

로더는 오류가 하나라도 있으면 아무것도 받아들이지 않습니다(fail closed). 알 수 없는 키, 철자가 틀린 동작, 형식이 잘못된 주소, 범위를 벗어난 숫자가 하나라도 있으면 파일 전체를 거부하고, 진단 메시지가 해당 규칙과 필드를 알려 주며, 데몬은 시작하지 않습니다. 알 수 없는 옵션이나 범위를 벗어난 값이 있는 명령줄도 같은 방식으로 거부됩니다. `rules`가 있으면 내장 테이블을 대체합니다. `ip_rules`는 수천 개의 접두사까지 확장되는 평판 트라이에 들어갑니다. `waf: false`는 파일에서 Gate 운영 구성을 선택합니다. 스키마에는 두 가지 키가 더 있습니다(제X부). `inspection`은 WAF 범주마다 `disabled`, `audit`, `enforce` 중 하나를 설정하고, 규칙의 `limits` 객체는 그 규칙에 자체 할당량을 줍니다. 규칙의 `ja4` 필드는 신뢰할 수 있는 인그레스가 전달한 JA4 지문과 정확히, 또는 `*`를 붙인 접두사로 일치시킵니다. 이 필드에는 `--ja4-header`가 필요합니다("인그레스에서 받는 JA4 지문" 참조). 경로 패턴은 정규형의 요청 경로와 일치시키므로(SID 0003), 패턴도 그 형식으로 써야 합니다.

영속 저장소와 클러스터

> **학습 목표** SQLite 또는 PostgreSQL 저장소를 켜고, SQL로 동적 정책과 차단을 추가하고, 인시던트 포렌식을 질의하고, 캠페인 군집화를 이해하고, PostgreSQL 데이터베이스 하나로 여러 노드를 실행합니다.

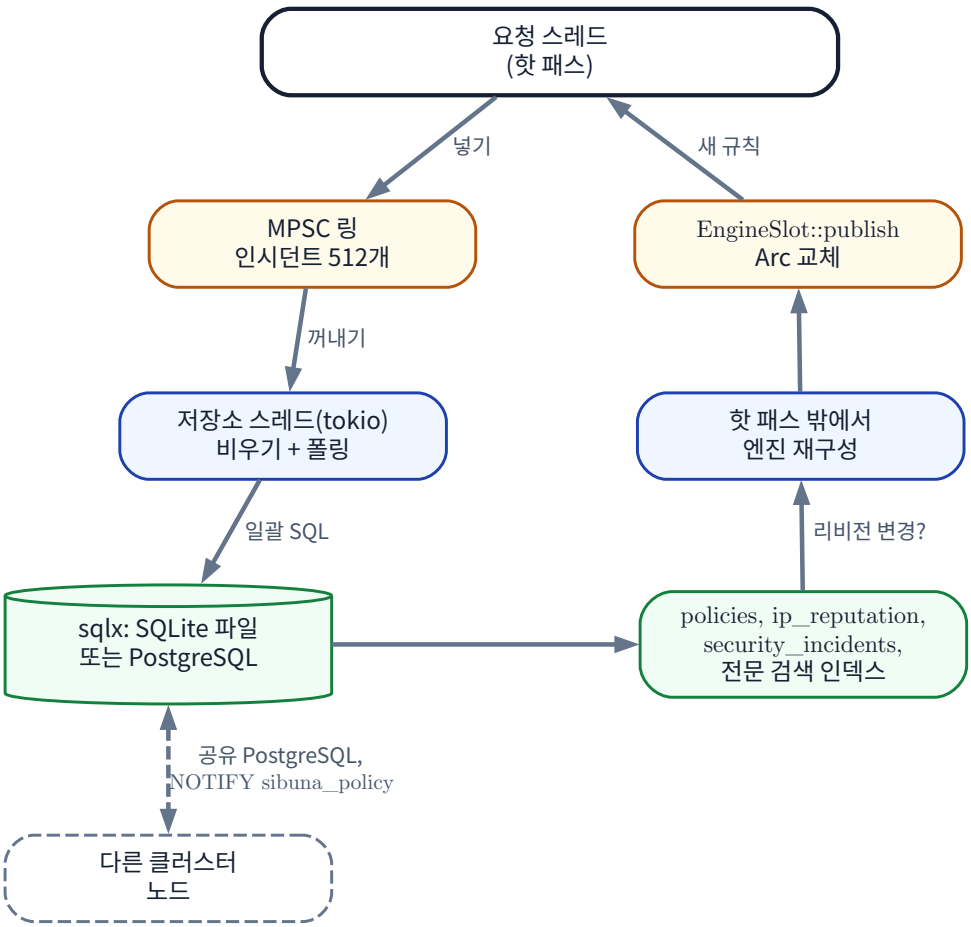


그림 25 저장소 아키텍처: 워커는 데이터베이스를 기다리며 멈추지 않습니다

`--data-dir /var/lib/sibuna`는 SQLite 파일 `/var/lib/sibuna/sibuna.db` (WAL 모드, `synchronous=NORMAL`, busy 타임아웃)를 열고 저장소 스레드를 시작합니다. `--database-url` (또는 `--database-url-file`)은 대신 PostgreSQL 데이터베이스에 연결합니다. 저장소 스레드는 단일 스레드 tokio 런타임과 작은 연결 풀을 소유하며, SQL을 실행하는 유일한 곳입니다. 시작할 때, 정책 리비전이 바뀔 때마다(`policies`, `ip_reputation`, `policy_inspection`에 쓰기가 일어날 때), 그리고 평판 항목이 만료될 때, 이 스레드는 정책 파일과 데이터베이스로 엔진을 다시 만들어 게시합니다. 요청은 SQL을 기다리지 않습니다. 재구성은 로드 전후에 리비전을 읽고, 그 사이에 쓰기 작업이 커밋되었으면 후보를 버립니다.

스키마

표 25 핵심 테이블

테이블	용도와 키
`policy_revision`	정책 리비전, 한 행. 세 정책 테이블의 트리거가 이 값을 올립니다.

테이블	용도와 키
policies	순서가 있는 동적 규칙, 키는 id. null 허용 ja4_pattern은 규칙의 JA4 조건을 최대 37바이트로 담습니다.
policy_inspection	범주별로 저장된 WAF 모드, 한 행.
ip_reputation	점수, 적중 수, 만료, 키는 주소 또는 접두사.
incident_receipts	각 노드의 영속 인시던트 영수증(마지막으로 커밋된 묶음 다음의 시퀀스), 키는 노드.
security_incidents	64개 성분 임베딩(256바이트)을 포함한 제한된 크기의 이벤트 기록 하나, 키는 node_id << 40 | sequence. null 허용 ja4는 요청의 JA4 지문을 정확히 36바이트로 담습니다.
incidents_fts	SQLite에서는 경로와 페이로드의 정규 토큰에 대한 FTS5 테이블입니다. PostgreSQL에서는 같은 토큰을 security_incidents의 tsvector 열 search에 담고 GIN 인덱스를 둡니다.
sibuna_schema	적용된 모든 마이그레이션, 키는 구성 요소와 버전.

핵심 스키마는 crates/storage/src/schema.rs이며, 방언마다 한 번씩 작성되어 있습니다. 콘솔이 있으면 구성 요소 console이 이름이 console_로 시작하는 테이블 33개를 추가합니다(SID 0013, crates/console-storage/src/schema/). 마이그레이션은 구성 요소마다 번호가 매겨져 바이너리에 컴파일됩니다. 핵심 마이그레이션 2는 JA4 열 두 개를 추가하며, 그 이전에 만든 데이터베이스는 시작할 때 마이그레이션됩니다. 대기 중인 마이그레이션은 모두 하나의 트랜잭션에서 실행되며, 노드는 어떤 구성 요소의 버전이 자신이 아는 것보다 새로운 데이터베이스를 거부합니다. 축약해 실은 스키마는 업그레이드 스크립트가 아닙니다. SQLite 테이블은 STRICT이고 CHECK 제약 조건이 Rust 타입을 그대로 반영하므로, 데몬이 표현할 수 없는 행은 데이터베이스가 거부합니다. 열거형 값은 소문자로 저장됩니다. 'DENY'를 직접 쓰면 거부됩니다.

운영 방법

sibuna.db에는 sqlite3 셸을 사용하고(WAL 모드 덕분에 데몬이 실행 중일 때도 읽을 수 있습니다), 클러스터 데이터베이스에는 psql을 사용하세요:

```
-- 동적 규칙을 추가합니다. 모든 노드가 다음 틱에 엔진을 다시 만듭니다.
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty, algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
VALUES ('p-checkout', 'protect-checkout', 10, '/checkout/*', 'challenge', 20, 'posw',
  '{"X-Api": "v2"}', '["10.0.0.0/8"]', 0, 1, unixepoch(), unixepoch());

-- 주소 하나를 클러스터 전체에서 하루 동안 차단합니다.
INSERT INTO ip_reputation (ip_or_cidr, reputation_score, banned_until,
  trigger_rule, hits, last_seen)
VALUES ('198.51.100.7', -100, unixepoch() + 86400, 'analyst', 1, unixepoch());

-- 포렌식: 기록된 페이로드를 전문 검색합니다.
SELECT s.id, s.client_ip, s.violation_category, s.path, s.campaign_id
FROM incidents_fts f JOIN security_incidents s ON s.id = f.rowid
WHERE incidents_fts MATCH 'union' ORDER BY rank LIMIT 20;
```

이 문장들은 SQLite용으로 작성되었습니다. PostgreSQL에서는 `enabled`가 `TRUE`이고, 시간은 `unixepoch()` 대신 `CAST(extract(epoch FROM now()) AS BIGINT)`이며, 검색은 `security_incidents`에서 `WHERE search @@ CAST('union' AS tsquery)`로 읽습니다. `path`와 `offending_payload`는 원시 요청 바이트(`BLOB` 또는 `BYTEA`)를 담습니다. 제약 조건을 어기는 직접 쓰기는 데이터베이스가 거부하며 노드에 도달하지 않습니다. PostgreSQL에서는 리비전 트리거가 `NOTIFY sibuna_policy`도 발행하므로 다른 노드가 즉시 틱을 실행합니다. 보장은 여전히 폴링이 제공합니다.

점수가 -50 이하이면 트라이의 `deny` 접두사가 되고, 50 이상이면 `allow`가 되며, `banned_until`은 차단 기간의 끝을 정합니다. 허니팟 적중은 -100 기록을 자동으로 삽입하므로, 공유 PostgreSQL 데이터베이스를 쓰면 한 노드에서 걸린 함정이 모든 노드에서 그 주소를 차단합니다.

트랜잭션 영수증으로 안전하게 재시도하기

저장소 스레드는 대기 중인 기록을 최대 32개까지 모아 SQL 트랜잭션 하나를 만듭니다. 이 트랜잭션은 인시던트를 삽입하고, 전문 검색 인덱스를 갱신하고, 허니팟 평판 변경을 적용합니다. 각 발급자(노드)에는 단조 증가하는 영수증, 즉 `incident_receipts`의 한 행이 있습니다. 데이터를 바꾸는 모든 문장은 이 영수증으로 보호됩니다. 트랜잭션은 영수증을 마지막에 올리며, 절대 되돌리지 않습니다.

풀이 예제: 확인 응답이 사라지는 경우 발급자 2가 시퀀스 101부터 132까지를 준비해 영수증 133으로 끝낸다고 합시다. 데이터베이스는 모든 기록과 영수증 133을 커밋하고, 확인 응답이 도착하기 전에 연결이 끊깁니다. 저장소 스레드는 준비된 묶음(식별자, 임베딩, 캠페인)을 유지한 채 다시 시도합니다. 영수증이 이미 133이므로 보호된 쓰기는 아무것도 하지 않습니다. 인시던트도, 허니팟 적중 수도 두 번 늘지 않습니다. 원래 트랜잭션이 롤백되었다면 이전 영수증이 남아 있으므로 재시도가 모든 쓰기를 한 번 적용합니다.

커밋에 실패하면 대기 중인 묶음을 메모리에 유지하고 `incident_write_failures`를 늘립니다. 틱은 그래도 정책 폴링을 시도합니다. 기록 512개짜리 큐가 가득 차면 `incidents_dropped`를 셉니다. 디스크를 기다리느라 HTTP 응답을 막는 일은 없습니다. 대기 중인 묶음은 기록 32개만큼의 공간을 더합니다. 커밋 전에 프로세스가 죽으면 큐에 있던 데이터를 잃을 수 있습니다. 이는 크기가 제한된 비동기 포렌식 경로이지, 큐에 넣는 순간 영속성이 보장되는 메시지 큐가 아닙니다.

연습 문제 9.1. 영수증 갱신을 데이터 트랜잭션 밖으로 옮겨 보세요. 평판 효과가 중복되는 크래시 시나리오 하나와 인시던트가 사라지는 크래시 시나리오 하나를 구성해 보세요.

캠페인 군집화

각 인시던트 페이로드는 바이트 트라이그램(숫자와 대소문자는 하나로 정규화)에 해싱 트릭을 적용해 64차원 단위 벡터로 임베딩하고, `security_incidents.embedding`에 256바이트로 저장합니다. 저장소 스레드는 모든 노드의 가장 최근 임베딩 16,384개를 메모리(4 MiB)에 두고, 틱마다 데이터베이스에서 새로 고칩니다. 삽입하기 전에 같은 묶음의 앞선 기록을 포함해 그중 가장 가까운 것을 찾습니다. 코사인 거리가 0.35 이하이면 새 인시던트는 그 인시던트의 `campaign_id`에 합류하고, 그렇지 않으면 자신의 식별자를 이름으로 하는 캠페인을 시작합니다. 이 구간보다 오래된 인시던트는 비교하지 않습니다. 유사도는 휴리스틱입니다. 회귀 테스트 예제는 관련된 SQL

페이로드를 한 그룹으로 묶지만, 모든 공격 쌍이 같은 캠페인에 속한다고 보장하지는 않습니다. 실행되는 모델도 없고 학습하는 것도 없습니다.

클러스터

클러스터는 PostgreSQL 데이터베이스 하나를 공유하는 노드의 집합입니다. postgres-tls 기능으로 빌드하고(cargo build --release -p sibuna --features postgres-tls, postgres를 포함합니다), 각 구성원을 데이터베이스와 고유한 노드 id로 시작합니다:

```
sibuna --database-url-file /etc/sibuna/database-url --cluster-node 1 \
       --secret-file /etc/sibuna/secret
```

모든 구성원은 같은 마스터 비밀 값과 1에서 8,388,607 사이의 고유한 --cluster-node를 사용해야 합니다. URL 파일은 postgres:// URL 하나를 담으며 소유자만 읽을 수 있어야 합니다. 배너는 URL을 절대 출력하지 않습니다. 쓰기는 PostgreSQL 주 서버로 갑니다. 각 노드의 저장소 스레드는 커밋된 리비전 변경을 (NOTIFY 또는 폴링으로) 감지하고 엔진을 다시 만듭니다.
노드는 네트워크를 거쳐 공유 데이터베이스에 접근하므로 그 연결을 암호화하고 검증합니다. 서버에는 자체 인증 기관이 서명한 인증서를 주고, URL에 그 인증 기관을 지정합니다: postgres://sibuna:...@db.internal:5432/sibuna?sslmode=verify-full&sslrootcert=/etc/sibuna/db-ca.pem. 서버에서는 pg_hba.conf의 hostssl 줄로만 TCP를 받습니다.
데몬은 시작할 때 데이터베이스 연결에 관한 줄 하나를 출력합니다. verify-full이면 인증서와 호스트 이름이 검증되었다고 알립니다. 그보다 약한 모드이면 경고와 힌트를 출력합니다. sslmode가 없으면 모드는 prefer입니다. 이때 연결은 서버가 TLS를 제공할 때만 암호화되며, 서버는 검증되지 않습니다. postgres-tls 없이 빌드한 데몬은 require, verify-ca, verify-full을 거부하고 빠진 기능의 이름을 알려 줍니다. 평문 연결로 물러서는 일은 절대 없습니다. 인증서가 거부되면 시작이 즉시 멈추며, 힌트는 sslrootcert를 확인하고 URL의 호스트가 서버 인증서에 있는 이름인지 확인하라고 안내합니다. sslrootcert는 우리 인증 기관을 시스템 신뢰 저장소에 추가할 뿐, 시스템의 인증 기관을 제거하지는 않습니다(SID 0011, SID 0012).
Sibuna는 자체 데이터베이스를 복제하지 않으며 리더를 선출하지도 않습니다. 복제, 장애 조치, 백업은 PostgreSQL의 몫입니다(스트리밍 복제, Patroni 또는 관리형 서비스). 콘솔의 노드 화면에는 리더가 없습니다. SQLite 노드에는 역할 single, PostgreSQL 노드에는 unknown을 보여 주며, quorum은 노드의 마지막 데이터베이스 왕복이 성공했다는 뜻일 뿐입니다. 데이터베이스에 연결할 수 없으면 마지막으로 게시된 엔진이 계속 서비스하고, 인시던트는 크기가 제한된 링에 쌓였다가 가득 차면 카운터를 올리며 버려지고, 저장소 스레드는 백오프하며 재시도합니다. 노드는 다시 연결될 때까지 마지막 정책을 유지합니다. Zaxonlite 데이터 디렉터리(SID 0005)는 읽지 않습니다. 정책은 sibuna console policies export와 sibuna console policies import로 옮기며, 포렌식 기록은 이전되지 않습니다.

클러스터 챌린지 라우팅

챌린지 키는 --cluster-node에 묶여 있으므로, 챌린지 발급과 검증은 같은 구성원에서 처리하세요. 토큰은 공유 시드를 사용하므로 다른 구성원에서도 동작합니다. 로컬 속도 할당량이 전역 할당량이 되지는 않으며, 프로세스를 재시작하면 사용 완료 집합이 지워집니다. 로컬 PostgreSQL 하나에 노드 세 개를 둔 경우, 직접 쓴 정책 행은 틱 한 번 안에 세 노드 모두에 도달하고, 한 노드

의 허니팟 차단이 다른 노드에도 도달하며, 인시던트 식별자는 절대 충돌하지 않습니다(crates/storage/tests/multi_node.rs, SID 0011 기준 2. SIBUNA_TEST_DATABASE_URL을 설정하면 실행됩니다).

패키징

> **학습 목표** 릴리스 패키지를 설치하고 검증하며, Sibuna를 서비스로, 컨테이너에서, 또는 Kubernetes에서 실행하고, 바이너리를 직접 빌드합니다.

GitHub의 모든 릴리스에는 플랫폼마다 같은 실행 파일이 들어 있습니다. 이 실행 파일은 릴리스 태그에서 기본 기능(SQLite 저장소와 콘솔)으로 빌드됩니다. PostgreSQL 클러스터에는 직접 빌드한 바이너리가 필요합니다(아래 참조).

플랫폼	릴리스 파일
Linux x86-64, ARM64	sibuna-linux-amd64.tar.gz와 sibuna-linux-arm64.tar.gz에는 어느 배포판에서나 실행되는 정적 musl 실행 파일이 들어 있습니다. sibuna_<version>-1_amd64.deb, sibuna_<version>-1_arm64.deb, sibuna-<version>-1.x86_64.rpm, sibuna-<version>-1.aarch64.rpm은 이 실행 파일을 systemd 서비스와 함께 설치하고, Arch Linux에서는 sibuna-bin-<version>-1-x86_64.pkg.tar.zst가 같은 일을 합니다.
macOS arm64, x86-64	sibuna-macos-arm64.tar.gz, sibuna-macos-amd64.tar.gz. macOS 12 이상용입니다.
Windows x86-64	sibuna-windows-amd64.zip에는 sibuna.exe가 들어 있습니다. Windows 10 또는 Server 2016 이상용입니다.
FreeBSD 15.1, OpenBSD 7.9 (amd64)	sibuna-<version>-freebsd-15.1-amd64.pkg와 sibuna-<version>.tgz.
Homebrew, Kubernetes, 컨테이너	sibuna.rb, helm-sibuna-<version>.tgz, 그리고 amd64와 arm64용 이미지 ghcr.io/insanai/sibuna:<version>. 이미지의 다이제스트는 IMAGE-DIGEST.txt에 있습니다.
소스	sibuna-<version>-source.tar.gz.

모든 아카이브와 패키지에는 LICENSE, NOTICE, LICENSES/ (실행 파일에 들어간 크레이트 목록인 THIRD-PARTY-RUST.txt 포함), README.md, SECURITY.md, sibuna.build.json이 들어 있습니다. 이 매니페스트는 버전, 커밋, Rust 타깃, 컴파일러 버전, 기능, 그리고 실행 파일과 두 WebAssembly 모듈의 SHA-256을 기록합니다. 릴리스 워크플로는 무엇이든 게시하기 전에 모든 실행 파일을 각자의 플랫폼에서 시작해 보고, Debian, Fedora, Arch Linux, FreeBSD, OpenBSD에서 패키지를 설치, 재설치, 제거해 봅니다.

릴리스 파일에는 서명이 없습니다. 설치하기 전에 내려받은 파일을 같은 릴리스의 SHA256SUMS와 대조해 확인합니다:

```
sha256sum --ignore-missing -c SHA256SUMS          # Linux
shasum -a 256 --ignore-missing -c SHA256SUMS      # macOS, FreeBSD
sha256 -C SHA256SUMS sibuna-1.0.0-beta.3.tgz      # OpenBSD
```

macOS에서 브라우저로 내려받은 파일에는 서명되지 않은 실행 파일을 막는 격리 속성이 붙습니다. 확인을 마친 뒤 `xattr -d com.apple.quarantine sibuna`로 이 속성을 제거합니다.

systemd 서비스

Debian, RPM, Arch 패키지는 `/usr/bin/sibuna`, 유닛 `sibuna.service`, 계정 `sibuna` (`systemd-sysusers`를 통해), 도우미 스크립트 `/usr/lib/sibuna/seed.sh`, 설정 파일 `/etc/sibuna/service.env`를 설치합니다. 설치만으로는 아무것도 시작되지 않습니다. 설정 파일에 원본 서버를 IP 리터럴로 지정하고 서비스를 시작합니다:

```
sudoedit /etc/sibuna/service.env      # SIBUNA_UPSTREAM_HOST, SIBUNA_UPSTREAM_PORT
sudo systemctl enable --now sibuna
curl -s http://127.0.0.1:8080/__sibuna/health
```

서비스는 `SIBUNA_HOST`와 `SIBUNA_PORT`로 달리 지정하지 않는 한 `127.0.0.1:8080`에서 수신합니다. `SIBUNA_OPTIONS`는 공백 기준으로 나눈 추가 옵션을 전달합니다. 예를 들면 `--mode forward_auth`나 `--console 127.0.0.1:19446`입니다. `seed.sh`는 매번 시작하기 전에 요청 허용 시드 `/var/lib/sibuna/admission.seed` (16진수 64자리, 모드 0600, 소유자 `sibuna`)가 없으면 만듭니다. 기존 시드는 절대 교체하지 않습니다. 새 시드는 모든 세션을 끝내기 때문입니다. 데몬은 SQLite 데이터베이스를 모드 0700 디렉터리인 `/var/lib/sibuna`에 둡니다. 유닛은 capability 없이 실행되며, 시스템을 읽기 전용으로 두고(`ProtectSystem=strict`), 비공개 `/tmp`와 장치를 쓰며, 새 권한 획득을 막습니다(no new privileges).

업그레이드는 설정 파일, 시드, 상태를 유지하며 서비스를 재시작하지 않습니다. 서비스는 백업한 뒤 직접 재시작합니다. 제거는 서비스를 중지하고 비활성화하지만 계정, 시드, 설정, 상태는 남겨 둡니다. 이것들은 호스트를 폐기할 때 직접 삭제합니다.

FreeBSD와 OpenBSD 패키지는 `/usr/local/bin/sibuna`와 그 문서만 설치합니다(`pkg add`, 또는 OpenBSD 미러 밖에서 받은 패키지라면 `pkg_add -D unsigned`). 서비스, 계정, 상태는 만들지 않습니다. 데몬은 비특권 사용자로, 전용 시드와 데이터 디렉터리를 두고 실행합니다.

Homebrew

`sibuna.rb`는 머신에 맞는 릴리스 아카이브(macOS 또는 Linux, arm64 또는 x86-64)를 설치합니다. 소스에서 빌드하지는 않습니다. 내장된 두 WebAssembly 모듈을 빌드하려면 `wasm32-unknown-unknown`용 표준 라이브러리가 필요한데, Homebrew의 Rust에는 이것이 없기 때문입니다. 아카이브에는 릴리스 워크플로가 이미 확인한 실행 파일이 들어 있습니다. Homebrew는 탭에서만 포뮬러를 설치하므로, 포뮬러를 로컬 탭에 넣습니다:

```
brew tap-new "$USER/local"
cp sibuna.rb "$(brew --repository)/Library/Taps/$USER/homebrew-local/Formula/"
brew install "$USER/local/sibuna"
```

포뮬러는 서비스를 시작하지 않습니다. 포뮬러의 주의 사항(caveats)은 시드를 만드는 방법을 보여 주고, 테스트는 같은 데이터 디렉터리에서 데몬을 두 번 시작합니다.

컨테이너와 Kubernetes

이미지 ghcr.io/insanai/sibuna:<version>는 distroless 기반 위에 정적 실행 파일을 담습니다. CA 인증서와 사용자 65532가 있고, 셸과 패키지 관리자는 없습니다. 진입점이 sibuna이므로 인수는 데몬 옵션입니다. 릴리스 워크플로는 이 이미지를 capability 없이 읽기 전용으로 실행하되, 시드는 읽기 전용으로 마운트하고 상태용 볼륨을 붙이며, 한 번 재시작해 봅니다:

```
docker run -d --read-only --cap-drop=ALL --security-opt=no-new-privileges \
  -v "$PWD/admission.seed:/run/admission.seed:ro" -v sibuna-state:/var/lib/sibuna \
  -p 8080:8080 ghcr.io/insanai/sibuna:1.0.0-beta.3 \
  --host 0.0.0.0 --secret-file /run/admission.seed --data-dir /var/lib/sibuna/data \
  --upstream-host 10.0.0.5 --upstream-port 3000
```

Helm 차트 helm-sibuna-<version>.tgz는 이 이미지의 Pod 하나를 실행합니다. 차트에는 직접 만든 Secret(secret.existingSecret, 키 admission.seed)이 필요하므로, 시드는 values나 템플릿에 절대 나타나지 않습니다. Pod는 Restricted Pod 보안 표준 아래에서 읽기 전용 루트 파일 시스템을 사용해 UID 65532로 실행됩니다. Recreate 업그레이드를 쓰는 레플리카 하나가 ReadWriteOnce 볼륨의 작성자를 하나로 유지하며, Sibuna는 이 볼륨의 data/에 SQLite 상태를 둡니다. 볼륨 클레임은 제거할 때도 남습니다. 프로브는 /__sibuna/health를 호출합니다. Service는 ClusterIP이고, 차트는 인그레스도 콘솔도 만들지 않으며, 차트의 NetworkPolicy는 networkPolicy.ingressFrom과 networkPolicy.egressTo가 클라이언트와 원본 서버를 허용할 때까지 모든 트래픽을 거부합니다. upstream.host는 IP 리터럴이어야 하며, extraArgs는 추가 데몬 옵션을 전달합니다. 워크플로는 Restricted 표준을 강제하는 kind 클러스터에서 차트를 설치, 업그레이드, 제거해 봅니다.

```
kubectl -n sibuna create secret generic sibuna-admission --from-file=admission.seed
helm install gateway ./helm-sibuna-1.0.0-beta.3.tgz --namespace sibuna \
  --set secret.existingSecret=sibuna-admission \
  --set upstream.host=10.0.0.5 --set upstream.port=3000
```

소스에서 빌드하기

우리는 Rust 1.100으로 빌드하고 테스트합니다. 최소 지원 버전은 여전히 Rust 1.94입니다. 우리는 “rustup toolchain install 1.100.0 –profile minimal –target wasm32-unknown-unknown“으로 릴리스 도구 모음을 설치하고, 아래 명령에서 “cargo +1.100.0“으로 선택합니다.

- cargo build --release -p sibuna는 기본 기능(SQLite 저장소와 콘솔)을 갖춘 target/release/sibuna를 만듭니다. sqlx는 SQLite를 번들로 포함하고, ring (rustls)은 빌드할 때 C 컴파일러가 필요합니다. 바이너리는 시스템 C 런타임을 동적으로 링크합니다. 빌드 스크립트는 WebAssembly 솔버와 콘솔 인터페이스도 빌드하며, 이를 위해 rustup target add wasm32-unknown-unknown이 필요합니다. SIBUNA_POW_WASM=<path>와 SIBUNA_CONSOLE_WASM=<path>를 지정하면 대신 미리 빌드한 모듈을 내장합니다.
- cargo build --release -p sibuna --no-default-features는 외부 크레이트가 전혀 없는 인메모리 방화벽을 만듭니다. 이때 --data-dir와 --database-url은 시작할 때 StorageDisabled로, sibuna console은 CONSOLEBUILD로 거부됩니다.
- cargo build --release -p sibuna --features postgres-tls는 데이터베이스와의 연결에 검증된 TLS를 쓰는 PostgreSQL 클러스터 지원을 추가합니다. OpenSSL은 필요하지 않습니다.

- 릴리스의 정적 Linux 실행 파일은 `musl-gcc`를 C 컴파일러로 지정하고 (`CC_x86_64_unknown_linux_musl=musl-gcc`) `cargo build --release --locked -p sibuna --target x86_64-unknown-linux-musl` (또는 `aarch64-unknown-linux-musl`)로 빌드한 것입니다.

릴리스 프로필은 fat LTO와 코드 생성 단위 하나를 사용합니다. 워크플로는 두 WebAssembly 모듈을 한 번만 빌드해 모든 플랫폼에 전달하므로, 한 릴리스의 모든 실행 파일은 바이트 단위로 동일한 모듈을 내장합니다. 의존성이 바뀌면 `cargo xtask licenses`로 `LICENSES/THIRD-PARTY-RUST.txt`를 다시 생성합니다.

Windows에서 데몬은 Ctrl+C, Ctrl+Break, 콘솔 닫기 이벤트에서 멈추며, Windows 서비스로는 실행되지 않습니다. Windows 파일에는 모드 비트가 없으므로, Sibuna는 다른 사용자가 읽을 수 있는 키, 자격 증명, 데이터베이스 URL 파일을 거부할 수 없습니다. 이런 파일은 ACL로 접근을 제한합니다. 콘솔의 메모리와 CPU 게이지는 `/proc`에서 읽으므로 다른 시스템에서는 사용할 수 없습니다.

연습 문제 9.2. 스테이징 네트워크(`10.20.0.0/16`)는 챌린지를 건너뛰고, `/admin/*`는 순차 작업 증명으로 작업 비트 20을 요구하며, `X-Partner-Key`로 식별되는 파트너 스크레이퍼는 10초당 요청 30개로 허용되도록 정책 파일과 `ip_reputation` 행을 작성하세요.
힌트: 데몬 플래그는 모든 요청이 가장 먼저 만나는 클라이언트별 제한을 설정합니다. 규칙은 `limits`에 자체 할당량을 둘 수 있습니다.

불변식 설명하기. `policies`에 행을 추가하면 재시작 없이, 그리고 어떤 요청도 기다리게 하지 않고 효과가 나타나는 이유를 저장소 스레드와 엔진 슬롯의 관점에서 운영자에게 설명해 보세요.

제X부

참고 자료와 진단

엔드포인트, 지표, 오류 목록, 정책 스키마, 저장소 테이블을 한곳에 모았습니다.

엔드포인트

학습 목표 `/__sibuna/` 아래의 모든 경로와 각 경로가 무엇을 반환하는지 알 수 있어야 합니다.

표 27 예약된 경로

엔드포인트	메서드	기능
`/__sibuna/challenge`	GET	중간 페이지
`/__sibuna/challenge.json?need=&path=`	GET	상태 없는 챌린지를 발급합니다. 챌린지 응답에 담긴 요구 티켓이 챌린지를 결정하며, 티켓이 없으면 보고된 URL을 정규형으로 평가합니다(8 KiB를 넘으면 `414`, 정규형이 없는 경로는 `400`). `{id, algorithm, difficulty, challenges, expires_at}`을 반환합니다
`/__sibuna/verify`	POST	`{"challenge_id", "nonce"}` 또는 `{"challenge_id", "proof"}`를 받습니다. `Set-Cookie`와 함께 `200`, 진단과 함께 `400`, 본문이 연결 버퍼보다 크면 `413`을 반환합니다
`/__sibuna/wasm/sibuna-pow.wasm`	GET	솔버 모듈, 14,744바이트(`benchmarks/results/primitives-latest.json`의 `wasm_bytes`), 1시간 동안 캐시 가능
`/__sibuna/worker.js`	GET	WASM 증명자와 JavaScript 증명자를 담은 Web Worker, 1시간 동안 캐시 가능
`/__sibuna/honeypot`	GET	호출자를 `--ban-seconds` 동안 차단하고 인시던트를 기록합니다. `403`을 반환합니다. 최상위 요청(`Sec-Fetch-Dest`가 없거나 `document`이고, 교차 사이트 `Sec-Fetch-Site`가 아닌 요청)만 차단하며, 그 밖의 요청은 `404`로 응답하고 횟수를 셉니다
`/__sibuna/health`	GET	엔진 이름, 버전, 프록시 모드, 증명 알고리즘을 담은 JSON 생존 확인(liveness) 상태

엔드포인트	메서드	기능
/__sibuna/metrics	GET	Prometheus 텍스트 형식(버전 0.0.4)

/__sibuna/ 아래의 그 밖의 경로는 404로 응답합니다. 발급과 검증은 별도의 --challenge-rate-limit 예산을 쓰며, 차단된 주소는 모든 경로에서 403으로 거부됩니다. 챌린지 응답에는 X-Sibuna-Status: CHALLENGE가 붙습니다. 허용된 요청은 업스트림으로 X-Sibuna-Status: PASS와 X-Sibuna-Rule을 전달합니다(forward-auth 응답에는 16진수 X-Sibuna-Rule-Hash도 붙습니다). HTML을 받아들이지 않는 클라이언트는 {"error":"challenge_required","challenge":"/__sibuna/challenge.json?need=<ticket>"}와 함께 401을 받습니다.
별도로 켜는 관리 리스너는 /console/을 제공합니다. 관리자 계정은 sibuna init-admin으로 로컬에서 초기화하고, 처음 로그인할 때 임시 비밀번호를 바꿉니다. /console/api/stats와 /console/ws에는 인증된 세션(키오스크 세션 포함)이 필요합니다. WebSocket은 통계, 이벤트, 노드, 정책, 챌린지, 감사 기록을 크기가 제한된 스냅숏, 변경분, 누락 복구와 함께 다중화합니다. 관리 피어는 /console/peer에 연결하며, 모든 API 경로는 /console/api/ 아래에 있습니다. forward-auth 모드에서는 원본 서버 오류 카운터를 관찰하지 않습니다. 콘솔의 워크플로, HTTPS 배포, 현재 SID 0007의 한계는 제IX부에서 설명합니다.

지표

다음 카운터를 이 순서대로 sibuna_<name>_total이라는 이름으로 Prometheus 텍스트 형식에 노출합니다.

표 28 카운터

카운터	증가 조건
incidents_persisted, incident_batches	저장소 스레드가 커밋을 확인한 레코드와 트랜잭션
incidents_dropped, incident_write_failures	링이 가득 차서 거부된 큐 삽입, 그리고 실패한 커밋 시도
requests	파싱된 요청이 디스패치에 도달함(예약된 경로 포함)
allowed, denied, challenged	정책 결정이 내려짐(허용, 거부, 챌린지 발급 또는 재발급)
challenges_issued	/__sibuna/challenge.json이 챌린지 레코드를 발급함
solutions_accepted, solutions_rejected	/__sibuna/verify가 증명을 받아들이거나 거부함
rate_limited	GCRA가 요청을 거부함(429): 요청 한도, 챌린지 예산, 또는 규칙의 할당량
banned	차단된 주소가 거부되었거나, 허니팟이 주소를 차단함
bans_issued	차단이 기록됨(허니팟 또는 규칙 할당량에 의한 차단)
proxied, upstream_errors	요청이 원본 서버로 전달되었거나, 원본 서버가 실패함(502)

카운터	증가 조건
parse_errors	형식이 잘못된 헤드, 반복된 Host, 정규형이 없는 대상(400), 또는 지원하지 않는 전송 코딩(501)이 거부됨
overloaded	--max-connections 또는 --max-connections-per-ip를 넘는 연결에 503으로 응답함
bans_refused	키가 들어갈 수 있는 모든 슬롯에 유효한 차단이 있어서 차단을 기록하지 못함
forwarded_invalid	신뢰하는 전달된 클라이언트 주소를 파싱하지 못해, 소켓 피어로 요청을 판정함
honeypot_refused	사람이 아니라 페이지가 요청했으므로 허니팟에 적중했지만 아무도 차단하지 않음
rate_evictions	제한기에 새로 도착한 클라이언트가 다른 활성 클라이언트의 셀을 밀어냄
ja4_present	--ja4-header를 지정하고 전달을 신뢰할 때, 요청에 JA4 지문을 담은 그 이름의 헤더가 정확히 하나 있었음
ja4_invalid	--ja4-header를 지정하고 전달을 신뢰할 때, 헤더 형식이 잘못되었거나 반복되어 지문 없이 요청을 판정함

대기 중인 배치는 보관되므로, 재시도는 레코드를 잃지 않고도 incident_write_failures를 늘릴 수 있습니다. 일정 구간 동안의 증가량을 감시하세요. 누적값만으로는 큐 깊이를 알 수 없습니다.

상태 코드

표 29 데이터 평면의 상태 코드

코드	상황
200	허용됨(forward-auth), 중간 페이지(챌린지가 필요한 HTML 탐색), 내부 경로
302	현재 프로토콜에서는 쓰지 않습니다. 검증은 200으로 응답하고 페이지를 다시 불러옵니다
400	형식이 잘못된 요청, 반복된 Host 필드, 정규형이 없는 대상, 잘못된 전달 메타데이터, 형식이 잘못된 본문, 또는 Elm 스타일 진단과 함께 거부된 풀이
401	HTML을 받아들이지 않는 클라이언트나 forward-auth 모드에서 챌린지가 필요함
403	정책 또는 WAF 거부, 차단된 주소, 허니팟
404	/__sibuna/ 아래의 알 수 없는 경로, 또는 최상위 요청이 아닌 허니팟 적중
413	풀이 본문이 64 KB 연결 버퍼보다 큼

코드	상황
414	보고된 챌린지 URL이 8 KiB를 넘음
417	지원하지 않는 요청 기대 조건. `100-continue`는 로컬에서 처리합니다
429	GCRA 한도 초과. `Retry-After`는 초 단위입니다
431	요청 헤드가 16 KB를 넘음
501	chunked가 아닌 코딩을 지정한 `Transfer-Encoding`
502	원본 서버에 연결할 수 없거나, 원본 응답 헤드의 형식이 잘못되었거나 16 KB를 넘음
503	연결 한도(`--max-connections` 또는 `--max-connections-per-ip`)에 도달했거나 노드가 드레이닝 중입니다. 응답 뒤에 소켓을 닫습니다

오류 목록

`INVALID COMMAND LINE`은 데몬이 인식하지 못하는 옵션, 문서화된 범위를 벗어난 값, 잘못되었거나 빠졌거나 중복된 모드 선택이 있으면 시작을 멈춥니다. 오류 블록은 옵션, 지정한 값, 기대하는 범위, 오류(`UnknownOption`, `MissingValue`, `InvalidValue`, `InvalidMode`, `MissingMode`, `DuplicateMode`, Zaxonlite 클러스터 옵션에 대한 `RemovedOption`, 서로 배타적인 두 저장소 옵션에 대한 `ConflictingOptions`, `--cluster-node` 없이 데이터베이스 URL을 준 경우의 `MissingOption`)를 밝히고 힌트로 끝납니다. 데이터베이스 URL은 `[redacted]`로 표시됩니다. `--mode reverse_proxy` 또는 `--mode forward_auth`를 한 번만 지정하세요. `-m`은 같은 뜻의 짧은 옵션입니다. 거부된 정책 파일은 오류에 맞는 제목의 블록(예: `INVALID POLICY ACTION`)에 파일, 오류 이름, 규칙, 필드, 값을 출력합니다. 어떤 지문과도 일치할 수 없는 `ja4` 패턴은 `INVALID POLICY JA4`(`InvalidJa4`)를 출력하며, 그런 패턴을 담은 콘솔 관리 문서도 같은 오류 이름으로 거부됩니다. 콘솔 옵션은 따로 검사합니다. `CONSOLE002`는 형식이 잘못되었거나 반복된 콘솔 옵션을, `CONSOLE001`은 규칙을 어기는 콘솔 설정(예: 저장소 없이 콘솔을 켠 경우의 `StorageRequired`)을 가리킵니다.

> **구현: `ErrorKind::explain`** 모든 도메인 오류를 경계선, 설명, `Hint:`로 대응시킵니다. `crates/core/src/errors.rs`(crate `sibuna-core`)에 있습니다.

표 30 챌린지와 세션 오류

오류	원인	힌트
`MalformedChallenge`	식별자가 올바른 형식의 챌린지 레코드가 아님	새 챌린지를 가져와 바꾸지 않고 제출하세요
`InvalidChallengeTag`	태그가 인증되지 않음. 이 클러스터가 발급하지 않았거나 수정됨	챌린지는 위조할 수 없습니다. 새 챌린지를 요청하세요

오류	원인	힌트
ChallengeExpired	챌린지 TTL보다 오래되었거나, 미래 시각에 발급됨	새 챌린지를 요청하세요
FingerprintMismatch	다른 주소나 User-Agent에서 제출됨	챌린지를 가져온 클라이언트에서 제출하세요
DifficultyNotMet	Hashcash nonce에 필요한 0비트가 부족함	nonce 탐색을 계속하세요
InvalidProof	순차 작업 증명이 커밋된 라벨을 개방하지 못함	발급된 깊이와 개방 수로 증명자를 끝까지 실행하세요
WrongSolutionType	PoSW 챌린지에 nonce를, 또는 Hashcash에 증명을 제출함	풀이 필드를 알고리즘에 맞추세요
DoubleSpendAttempt	챌린지가 이미 사용됨	챌린지는 한 번만 쓸 수 있습니다
StoreFull	사용 완료 집합이 가득 참	챌린지 TTL을 낮추거나 --spent-tags를 늘리세요
InvalidTokenSignature	쿠키 태그나 서명 검증이 실패함	중간 페이지를 통해 다시 인증하세요
TokenExpired	쿠키가 만료됨	다시 인증하세요
TokenBoundAddressMismatch	쿠키가 다른 클라이언트 신원에서 제시됨	쿠키는 공유할 수 없습니다

콘솔 API는 실패를 {"error":"<CODE>","hint":"<sentence>"}로 응답합니다. 힌트는 인터페이스가 보여 주는 문장입니다(crates/console/src/http/reply.rs).

표 31 운영자와 관련된 콘솔 실패 코드

코드	상태	의미
CONSOLE002	400	요청의 형식이 잘못되었거나, 너무 크거나, 그 형태로는 허용되지 않음
CONSOLE003	503 / 429	저장소 작업 큐가 가득 찼거나(작업 32개), 비밀번호 검증기가 바쁨
CONSOLEQUORUM	503	데이터베이스가 요청 대기 시간 10초 안에 응답하지 않음. 결과는 알 수 없으며, 데이터 평면은 마지막으로 적용된 정책으로 계속 동작함
CONSOLE004	503	서비스가 응답할 수 없음: 이 빌드에서 제공되지 않거나, 중지 중이거나, 내부 실패

코드	상태	의미
CONSOLENODE	400 / 409 / 503	노드 명령이 거부됨: 다른 노드를 지정했거나(400), 부팅 또는 제어 리비전이 오래되었거나(409), 다른 명령이 실행 중일 때 도착했거나 완료를 기록할 수 없었음(503). 재시도하기 전에 작업 영수증을 확인하세요
CONSOLEMUTATION	429	한 세션에서 1분에 관리 변경이 60건을 넘음

정책 스키마

```
{
  "default_action": "ALLOW" | "DENY" | "CHALLENGE",
  "waf": true | false,
  "inspection": { "path_traversal" | "sqli" | "xss" | "rce": "disabled" | "audit" | "enforce" },
  "thresholds": { "challenge_at": int, "deny_at": int, "bits_step": int },
  "ip_rules": { "<cidr>": "ALLOW" | "DENY" | "CHALLENGE", ... },
  "rules": [                                                    // 최대 128개
    {
      "name": "<kebab-case>",                                   // 출력 가능한 바이트 1~128개
      "path" | "path_regex": "<pattern>",                       // 최대 512바이트
      "user_agent" | "user_agent_regex": "<pattern>",
      "headers" | "headers_regex": { "<Header>": "<pattern>" }, // 최대 4개
      "remote_addresses" | "cidrs": ["<cidr>", ...],            // 최대 8개, IPv4 또는 IPv6
      "action": "ALLOW" | "DENY" | "CHALLENGE" | "WEIGH",
      "weight": int,                                            // WEIGH 전용, -1000..1000
      "challenge": { "difficulty": <work bits>, "algorithm": "hashcash" | "posw" },
      "limits": { "rate": int, "window_seconds": int, "ban_seconds": int },
      "ja4": "<fingerprint>" | "<prefix>*"                      // 36바이트, 또는 접두사와 *
    }
  ]
}
```

challenge_at과 deny_at은 -1000에서 1000 사이이며 deny_at은 challenge_at 이상이어야 하고, bits_step은 1에서 1000 사이입니다. challenge 블록은 CHALLENGE 규칙에만 쓸 수 있으며 작업 비트는 1에서 64입니다. limits는 규칙에 자체 할당량을 줍니다. rate 1에서 1,000,000과 window_seconds 1에서 86,400은 필수이고, ban_seconds(최대 86,400)는 선택이며, WEIGH 규칙에는 한도를 둘 수 없습니다. ja4는 정확히 같아야 하는 전체 JA4 지문이거나, 지문이 그것으로 시작해야 하는 접두사 뒤에 * 를 붙인 것입니다. * 하나만 쓰면 모든 지문과 일치하고, 지문이 없는 요청은 어떤 ja4 규칙과도 일치하지 않습니다. 어떤 지문과도 일치할 수 없는 패턴은 InvalidJa4로 거부됩니다(제IX부, 인그레스에서 받는 JA4 지문). ip_rules의 평판 트라이는 노드를 최대 8,192개 담습니다. 각 규칙은 조건마다 한 가지 표기만 쓸 수 있습니다.

패턴 문법: .* 또는 *는 모든 것과 일치합니다. ^...$는 정확한 경로를 고정합니다. 끝의 *, /*, .* 는 접두사입니다. /로 시작하는 패턴은 정확한 경로입니다. 그 밖의 것은 대소문자를 구별하지 않는 부분 문자열입니다. 경로 패턴은 정규형 경로(세그먼트 사이에 슬래시 하나, 점 세그먼트 없음, 예약되지 않은 문자는 이스케이프 해제, 그 밖의 이스케이프는 대문자 16진수)와 대조하므로, ;, %2F, %5C, \를 담은 패턴은 아무것과도 일치하지 않으며, 경로 매칭은 대소문자를 구별합니다(SID 0003).

저장소 테이블

policy_revision, policies(최대 37바이트의 null 허용 ja4_pattern 조건 포함), policy_inspection, ip_reputation, incident_receipts, security_incidents(256바이트 임베딩과 null 허용 36바이트 ja4 지문 포함), 전문 검색 색인(SQLite에서는 정규 토큰에 대한 FTS5인 incidents_fts, PostgreSQL에서는 GIN 색인이 있는 tsvector 열 search), 그리고 sibuna_schema입니다. 콘솔은 console_* 테이블 33개를 더합니다. 저장되는 동작, 알고리즘, 모드 이름은 소문자입니다. 인시던트 id는 node_id << 40 | sequence로, 조정 없이도 클러스터 전체에서 유일합니다. 노드 id는 1부터 8,388,607까지입니다.

빌드 대상

표 32 빌드와 저장소 명령

명령	결과
cargo build --release -p sibuna	SQLite 저장소와 콘솔을 포함한 데몬(기본 기능)
cargo build --release -p sibuna --no-default-features	저장소와 콘솔 없이, 외부 crate도 없이 메모리에서만 동작하는 데몬
cargo build --release -p sibuna --features postgres-tls	공유 PostgreSQL 데이터베이스에도 검증된 TLS로 연결하는 데몬(postgres 기능만 켜면 TLS 없이 연결)
cargo test	단위 테스트, 실행 중인 데몬에 대한 통합 테스트, 저장소와 콘솔 테스트(SIBUNA_TEST_DATABASE_URL을 설정하면 PostgreSQL 테스트 포함)
cargo clippy --all-targets, cargo fmt --check	린트와 포맷(max_width = 99)
cargo xtask check	70줄 함수, 1,408줄 파일, 99/108열 줄 길이 한도. 탭 금지
cargo xtask browser	헤드리스 Chromium에서 실행하는 검사(sibuna-browser)
cargo xtask bench <primitives|product|latency>	릴리스 모드 벤치마크. 결과는 benchmarks/results/<command>-latest.json에 저장
cargo build -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm	브라우저 모듈만 빌드(콘솔 인터페이스는 -p sibuna-console-ui)
cargo run -p sibuna-geoip --bin geoip-snapshot -- ...	국가 데이터를 검증하고 내장할 수 있는 GeoIP 스냅숏을 생성
guidedog build / guidedog dsc	이 책 / SID 기록

데몬의 빌드 스크립트(cmd/sibuna/build.rs)는 두 WebAssembly 모듈을 wasm 프로파일로 워크스페이스의 대상 디렉터리에 빌드하며, 콘솔 인터페이스가 768 KiB를 넘으면 빌드를 실패시킵니다.

wasm32-unknown-unknown 대상이 없는 빌드에서는 SIBUNA_POW_WASM과 SIBUNA_CONSOLE_WASM으로 미리 빌드한 모듈을 대신 지정합니다. SIBUNA_GEOIP_DATA는 내장할 GeoIP 스냅숏을 지정합니다. 이 스냅숏은 빌드할 때 디코딩되며, 콘솔이 없는 빌드는 이를 거부합니다(GEOIP001).

연습 문제 10.1. 클라이언트가 풀이 도중 Wi-Fi에서 셀룰러로 바꾼 뒤 CLIENT FINGERPRINT MISMATCH 제목과 함께 400을 받습니다. 토큰과 챌린지 형식으로 원인을 설명하고, 중간 페이지가 매끄럽게 회복하도록 하는 가장 작은 변경을 제안하세요.

불변식 설명하기. 책을 보지 않고, Sibuna 앞의 리버스 프록시가 원본 서버가 아니라 데몬으로 보내야 하는 엔드포인트를 나열하고, 각각이 왜 필요한지 말하세요.

제X I부

빠른 참조 카드

터미널 옆에 두고 보세요. 빌드, 실행, 설정, 라우팅, 관찰, 진단을 다룹니다. 전체 표는 제X부에 있고, 이 카드에는 운영자가 실제로 입력하는 내용을 담았습니다.

빌드

```
cargo build --release -p sibuna                          # 데몬 + SQLite + 콘솔
cargo build --release -p sibuna --no-default-features    # 메모리 전용, 외부 crate 없음
cargo build --release -p sibuna --features postgres-tls  # 공유 PostgreSQL 클러스터
rustup target add wasm32-unknown-unknown   # 또는 SIBUNA_POW_WASM / SIBUNA_CONSOLE_WASM 설정
cargo test            # 단위, 실행 중인 데몬, 저장소, 콘솔 테스트
cargo clippy --all-targets && cargo fmt --check
cargo xtask check     # 70줄 / 1,408줄 / 99열 검사
cargo xtask browser   # 헤드리스 Chromium 검사
cargo xtask bench product   # wrk 부하 아래의 전체 제품 -> benchmarks/results/
guidedog build        # 이 책 (guidedog dsc: SID 기록)
```

실행

```
# Shield(기본값): PoW + WAF, :3000으로 프록시
sibuna -p 8080 -u 3000 -s /etc/sibuna/secret
# Gate: PoW만
sibuna --gate -p 8080 -u 3000 -s /etc/sibuna/secret
# Nginx/Caddy 뒤의 forward-auth (X-Forwarded-For를 신뢰)
sibuna -m forward_auth --host 127.0.0.1 -p 8080 -s /etc/sibuna/secret
# Edge: 영속 정책, 평판, 포렌식을 <dir>/sibuna.db에 저장
sibuna -D /var/lib/sibuna -s /etc/sibuna/secret
# 신뢰할 수 있는 인그레스가 보내는 JA4 (Cloudflare 헤더; nginx 모듈: x-ja4; ATS: ja4)
sibuna --ja4-header cf-ja4 --trust-forwarded -p 8080 -u 3000 -s /etc/sibuna/secret
# 클러스터 구성원: 공유 PostgreSQL, 고유한 노드 id
sibuna --database-url-file /etc/sibuna/database-url --cluster-node 1 -s /etc/sibuna/secret
head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret
```

콘솔(선택형 미리 보기)

```
# 데몬을 멈춘 상태에서 초기화한 다음 콘솔을 시작합니다
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
```

`http://127.0.0.1:19446/console/`을 열고 임시 비밀번호를 바꾸세요. `--console-location 1.3521,103.8198`을 추가하면 지구본에 노드가 표시됩니다. 원격 접근에는 HTTPS 프록시, 명시적 origin, 신뢰

할 프록시 CIDR이 필요합니다. 제IX부의 절차를 따르세요. SID 0007은 Committed 상태입니다. 콘솔은 헤드리스 Chromium에서 브라우저 워크플로 테스트를 통과합니다(SID 0010, 기준 5). 공식 성능 격리 관문은 아직 통과하지 못한 미결 항목입니다(제VIII부).

먼저 알아야 할 플래그

플래그	기본값	의미
`-d, --difficulty`	16	작업 비트(PoSW 깊이 = 비트 - 3)
`-a, --algorithm`	posw	`posw` 또는 `hashcash`
`--posw-challenges`	16	증명당 개방 수
`--token-ttl`	86400	세션 유효 시간(초)
`--challenge-ttl`	300	챌린지 유효 시간(초)
`--token-scheme`	mac	`mac` 또는 `ed25519`
`--rate-limit / --rate-window`	100 / 10	윈도당 GCRA 버스트
`--ban-seconds`	3600	허니팟 차단
`-w, --workers`	CPU 수	accept 스레드
`--max-connections`	1024	초과 시 `503`
`--max-connections-per-ip`	64	피어 주소별(IPv6: `--ipv6-prefix`별), 초과 시 `503`
`--trusted-hops`	1	클라이언트 = 오른쪽에서 n번째 `X-Forwarded-For` 항목
`--ja4-header`	꺼짐	인그레스의 JA4 지문을 담은 헤더. 신뢰할 때만 읽음
`--idle-timeout`	15	유휴 연결 회수 시간(초)
`-P, --policy-file`	없음	JSON 정책
`--trust-forwarded`	꺼짐	forward-auth에서 켜짐
`--secure-cookie`	꺼짐	`Secure` 추가

`/__sibuna/` 아래의 엔드포인트

경로	응답
`challenge`	중간 페이지 HTML
`challenge.json?need=&path=`	`{id, algorithm, difficulty, challenges, expires_at}`

경로	응답
verify(POST)	{challenge_id, nonce | proof} -> 200 + Set-Cookie, 아니면 400
wasm/sibuna-pow.wasm	15,181바이트 솔버
worker.js	WASM 증명자와 JS 증명자를 갖춘 워커
honeypot	호출자를 차단하고 403, 하위 리소스나 교차 사이트 가져오기에는 404
health	{"status":"ok",...}
metrics	Prometheus 텍스트

이 경로는 모두 원본 서버가 아니라 데몬으로 라우팅하세요.

상태 코드

코드	상황
200	허용(forward-auth), 중간 페이지, 내부 경로
400	잘못된 요청 또는 대상, 거부된 풀이(진단 본문)
401	챌린지 필요: HTML이 아닌 클라이언트 또는 forward-auth
403	정책 또는 WAF 거부, 차단, 허니팟
404	알 수 없는 /__sibuna/ 경로, 거부된 허니팟 적중
413	64 KB 버퍼를 넘는 풀이 본문
414	8 KiB를 넘는 보고된 챌린지 URL
417	지원하지 않는 요청 기대(Expect)
429	GCRA 제한, Retry-After 초
431	16 KB를 넘는 헤드
501	chunked가 아닌 전송 코딩
502	원본 서버에 연결할 수 없거나 응답이 잘못됨
503	연결 한도 도달 또는 노드가 드레이닝 중

헤더

- 업스트림*(프록시)*: X-Forwarded-For(소켓 피어, 신뢰하는 경우 받은 체인 뒤에 추가), X-Forwarded-Proto, X-Real-IP(판정한 클라이언트), X-Sibuna-Status: PASS, X-Sibuna-Rule: <rule>. 홉별(hop-by-hop) 필드와 그 밖의 전달 필드는 제거하며, 밑줄로 쓴 표기도 제거합니다.

- *forward-auth* 응답: `X-Sibuna-Status`, `X-Sibuna-Rule`, `X-Sibuna-Rule-Hash`.
- 챌린지 응답: `X-Sibuna-Status: CHALLENGE`.
- 쿠키: `__sibuna_token=<75 chars, 139 with ed25519>; Path=/; Max-Age=<ttl>; HttpOnly; SameSite=Lax[; Secure]`

정책 파일 골격

```
{ "default_action": "CHALLENGE", "waf": true,
  "thresholds": {"challenge_at": 10, "deny_at": 40, "bits_step": 5},
  "ip_rules": {"10.0.0.0/8": "ALLOW", "2001:db8::/32": "DENY"},
  "rules": [
    {"name": "api", "path": "/api/*", "headers": {"X-Api-Key": ".*"}, "action": "ALLOW"},
    {"name": "checkout", "path": "/checkout/*", "action": "CHALLENGE",
     "challenge": {"difficulty": 20, "algorithm": "posw"}},
    {"name": "login", "path": "^/login$", "action": "CHALLENGE",
     "limits": {"rate": 10, "window_seconds": 60, "ban_seconds": 600}},
    {"name": "old-tls", "ja4": "t12*", "action": "CHALLENGE"},
    {"name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight": 30} ] }
```

JA4: 전체 지문, 또는 접두사 뒤에 *. 지문이 없으면 일치하지 않습니다. 패턴: * 또는 .*는 모두 일치, ^…$는 정확히 일치, 끝의 *는 접두사, 앞의 /는 정확한 경로, 그 밖에는 대소문자를 구별하지 않는 부분 문자열입니다. 경로는 정규형으로 비교합니다. 순서: WAF -> 트라이 deny/allow -> 규칙 -> 점수 -> 우회 -> 트라이 challenge -> 봇 -> 기본값.

SQL로 보는 저장소(Edge)

```
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty, algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
  VALUES ('p1', 'protect', 10, '/checkout/*', 'challenge', 20, 'posw', '{}', '[]', 0, 1,
  unixepoch(), unixepoch());
INSERT INTO ip_reputation (ip_or_cidr, reputation_score, banned_until, trigger_rule, hits,
last_seen)
  VALUES ('198.51.100.7', -100, unixepoch() + 86400, 'analyst', 1, unixepoch());
SELECT s.client_ip, s.violation_category, s.path, s.campaign_id
  FROM incidents_fts f JOIN security_incidents s ON s.id = f.rowid
  WHERE incidents_fts MATCH 'union' ORDER BY rank LIMIT 20;
```

SQLite(`sqlite3 <dir>/sibuna.db`). 동작은 소문자로 씁니다. 점수 ≤ -50 -> deny 접두사, ≥ 50 -> allow. 클러스터: 모든 노드에 같은 비밀 값, PostgreSQL 데이터베이스 하나, `--features postgres-tls`, 1에서 8,388,607 사이의 고유한 N으로 `--database-url-file <file> --cluster-node N`. 멤버 목록, 리더, 클러스터 TLS는 없습니다(SID 0011).

진단

제목	의미
MALFORMED CHALLENGE	id가 수정되었거나 잘렸습니다. 새 챌린지를 받으세요
INVALID CHALLENGE TAG	이 시드나 노드가 발급하지 않았습니다
CHALLENGE EXPIRED	TTL보다 오래되었거나 시계 오차가 60 s 초과
CLIENT FINGERPRINT MISMATCH	주소나 User-Agent가 바뀌었습니다
DIFFICULTY NOT MET	nonce에 0비트가 부족합니다
INVALID PROOF OF SEQUENTIAL WORK	개방이 루트에 닿지 않습니다
WRONG SOLUTION TYPE	PoSW에 nonce, 또는 Hashcash에 증명을 보냈습니다
DOUBLE SPEND PREVENTED	이미 사용한 챌린지입니다
STORE FULL	사용 완료 집합이 포화되었습니다. TTL을 낮추거나 --spent-tags를 늘리세요
NETWORK BINDING MISMATCH	쿠키가 다른 클라이언트로 복사되었습니다
INVALID COMMAND LINE	알 수 없는 옵션, 잘못된 값, 또는 제거된 Zaxonlite 옵션

경보를 걸 지표: denied, rate_limited, overloaded, upstream_errors, incidents_dropped, incident_write_failures, bans_refused, forwarded_invalid(모두 sibuna_*_total).

기억할 숫자

챌린지 id 70자(36바이트 페이로드 + 16바이트 태그), 토큰 75자(40 + 16), Ed25519에서는 139자, 증명 바이트 $32(1 + t(n + 1))$, PoSW 깊이 = 비트 - 3, 범위 4에서 24, 요청 버퍼 64 KB, 헤드 한도 16 KB, 본문 검사 8 KB, 사용 완료 집합 태그 1,048,576개(--spent-tags), 속도 셀 테이블당 262,144개(--rate-table-cells), 차단 슬롯 65,536개(--ban-table-slots), 클라이언트 주소당 연결 64개, 인시던트 링 512, 배치 32, 가장 최근 임베딩 16,384개에 대한 캠페인 거리 0.35, 연결당 keep-alive 요청 4,096개, 원본 풀 소켓 256개, 저장소 작업 큐 32.

인그레스 라우팅

제IX부 "인그레스 뒤의 forward-auth"의 Nginx 또는 Caddy 설정 예를 사용하세요. /__sibuna/*는 직접 라우팅하고, 애플리케이션 요청은 원래 URI, 메서드, 신뢰하는 클라이언트 주소로 인가합니다. 들어오는 Sibuna 감사 헤더는 인가 결과로 교체합니다. Nginx에서는 챌린지, 429, 인가 서버를 쓸 수 없는 경우를 명시적으로 처리해야 합니다.
reverse_proxy는 허용된 HTTP/1.1 본문과 WebSocket을 전달합니다. forward_auth는 요청 허용만 결정하고 전달은 인그레스가 맡습니다. 인가 요청에 실리지 않은 본문과 원본 응답은 Sibuna가

검사하거나 집계할 수 없습니다. TLS는 인그레스에서 종료되며, 네이티브 HTTP/2는 뒤로 미뤘습니다. 업로드는 Content-Length나 chunked 코딩을 사용합니다. 검사는 최대 8 KiB 접두사만 보며, 파일 바이트는 내용을 해석하지 않습니다.

일부 연습 문제 해답

이 해답은 논증을 확인하기 위한 것이며 연습 문제를 대신하지 않습니다. 결과가 다르면 먼저 가정을 비교합니다. 시간 구간의 끝점, 무엇을 작업으로 세는지, 쓰기가 커밋되었는지가 흔히 어긋나는 원인입니다.

제I부: 비용과 확률

1.1. 시행 백만 회를 요청 100개로 나누면 요청당 시행 10,000회입니다. 전체 요청 수가 100으로 유지되면 세션을 공유해도 이 계산은 바뀌지 않습니다. 실제 시스템은 두 번째 프로세스가 지문과 라우팅 바인딩을 만족할 수 있는지 판단해야 합니다. 이 조건은 분할 상환 식 밖에 있습니다.
1.2. 초과분은 초당 $1000 - 600 = 400$개 레코드입니다. 레코드 512개짜리 빈 큐는 유체 모델에서 $512/400 = 1.28$초를 벌어 줍니다. 실제 생산자는 몰려서 도착하고 소비자는 배치로 커밋하므로, 첫 레코드가 버려지는 정확한 시점은 도착 시각과 커밋 시각에 달려 있습니다.

제II부: 선행 연구

2.1. 한 소스에서 두 번 컴파일한 결과는 명제, nonce 인코딩, 트리 모양, 개방 배치에 관해 서로 어긋날 수 없습니다. 어긋나려면 컴파일러 버그가 있어야 합니다. 그러나 구성이 건전한지(암호학적 검토), 브라우저 런타임이 이를 올바르게 실행하는지(WebAssembly 증명자와 JavaScript 증명자의 바이트 단위 비교가 이를 다룹니다), 매개변수가 보정되었는지(측정)에 대해서는 아무것도 말해 주지 않습니다.
2.2. (a) Sibuna Gate 또는 Anubis, 바이너리 하나. (b) Sibuna Shield, SafeLine, 또는 Cloudflare(Pro 요금제 이상). (c) 공유 PostgreSQL 데이터베이스(`--database-url`)를 쓰는 Sibuna Edge, 공유 Valkey 저장소를 쓰는 Anubis(상태용이며 차단용은 아님), 또는 Cloudflare. Docker 호스트나 구독 없이 세 조건을 모두 만족하는 단일 제품은 노드들이 PostgreSQL 데이터베이스 하나를 공유하는(`--database-url`과 `--cluster-node`) Sibuna Shield뿐입니다. "실행 형태", "검사", "다중 노드" 행이 이를 결정합니다.

제III부: 암호학

3.0. 꼬리 추정값은 $e^{-4.6} \approx 0.010$입니다. nonce 공간을 나누면 시행을 마치는 속도가 바뀔 뿐, 각 독립 시행의 성공 확률은 바뀌지 않습니다. 중복 시행이나 조정 작업 때문에 실제 병렬 구현은 효율이 더 낮을 수 있습니다.
3.1. 각 커밋먼트가 확률 q로 성공하면, g개가 모두 실패할 확률은 $(1-q)^g$입니다. 적어도 하나가 성공할 확률은 $1-(1-q)^g$입니다. 이 공식은 커밋먼트를 만드는 비용, 질의 예산, 시행 사이의 의존성, 특정 수락 확률을 가진 커밋먼트를 찾는 데 필요한 작업을 세지 않습니다.
3.2. 깊이 17, 개방 16개에서 증명은 $32(1 + 16 \times 18) = 9248$바이트입니다. 개방이 32개이면 $32(1 + 32 \times 18) = 18464$바이트입니다. 개방 바이트는 두 배가 되지만, 32바이트 루트는 여전히 한 번만 보냅니다.

3.3. 작업 수준이 없으면 값싼 경로에서 얻은 세션이 더 많은 작업을 요구하는 비싼 경로의 요청도 허용하게 됩니다. `timestamp`가 없으면 토큰을 "미래에" 발급해 정책보다 오래 살아남게 할 수 있습니다. `expiry`가 없으면 토큰은 결코 만료되지 않습니다. `rule_hash`가 없으면 업스트림은 어떤 규칙이 클라이언트를 허용했는지 알 수 없습니다. `fingerprint`가 없으면 다른 기계로 복사한 쿠키가 받아들여집니다. 태그는 다섯 가지를 모두 덮어야 합니다. 태그 밖에 남은 필드는 마음대로 고칠 수 있고, 검증자는 이를 알아챌 수 없기 때문입니다.

제IV부: 프로토콜

4.1. 두 브라우저는 주소와 User-Agent를 공유하므로 지문, 속도 제한 셀, 차단 슬롯을 공유합니다. 챌린지(각자 자신의 nonce를 가진 자신의 레코드를 가져옵니다), 사용 완료 집합 항목, 쿠키는 공유하지 않습니다. 다른 브라우저의 쿠키를 재사용하려면 NAT에 있는 공격자는 다른 기계에서 쿠키를 읽어 내야 합니다. 그러면 지문이 일치하게 되므로, 공유 주소에서는 바인딩보다 `HttpOnly`와 `SameSite`가 더 중요합니다.

제V부: 상태와 순서

5.1. 헤드와 본문의 첫 부분(64 KB 안에서 헤드 뒤에 들어가는 만큼)은 연결 버퍼에 있으며 `write_all(request.body)`로 보냅니다. 나머지 바이트는 결코 한꺼번에 버퍼에 들어오지 않습니다. `relay_body`가 클라이언트 리더에서 16 KB씩 읽어 각 조각을 원본 서버에 씁니다. 헤드를 먼저 보내야 하는 이유는 원본 서버가 본문보다 먼저 `Content-Length`를 알아야 하고, 감사 헤더가 헤드의 일부이기 때문입니다.

5.2. 방출 간격은 100 ms이고 버스트 허용치는 9,900 ms입니다. 유휴 상태의 클라이언트는 요청 100개를 즉시 보낸 뒤 100 ms마다 하나씩 보낼 수 있습니다. 3,000 ms 끝점을 포함하면 상한은 130입니다. 고정된 10초 카운터는 첫 창에서 100개를 허용하고, 창이 넘어가는 순간 100개를 더 허용합니다. 슬라이딩 로그는 어느 10초 동안이든 정확히 100개를 허용하지만, 클라이언트마다 타임스탬프 100개를 저장해야 합니다.

제VI부: 알고리즘

6.1. 이름 속의 아포스트로피와 `order`라는 평범한 매개변수는 SQL 표현식을 입증하지 않습니다. 부분 문자열 탐지기는 둘 중 어느 것이든 구문으로 오인할 수 있습니다. 구조적 탐지기는 연산자와 리터럴 관계를 포함한 추가 증거를 찾습니다. 그래도 이는 상한이 있는 검사이며, 모든 애플리케이션 언어를 위한 파서가 아닙니다.

6.2. LDAP 필터는 괄호, `|`, `&`, `!`, `=`로 만듭니다. 탐지기에는 `(`와 `)`를 위한 바이트 클래스가 새로 필요하며, `|`와 `&`에는 기존의 `=` 클래스와 셸 구분자 클래스를 재사용할 수 있습니다. SQL 탐지기가 따옴표나 항진식을 요구하는 것과 같은 이유로, 이 탐지기는 괄호 하나가 아니라 `)(`나 `(|(` 같은 연속에 반응해야 합니다.

제VII부: 브라우저 엔진

7.1. 기본값인 16 작업 비트(깊이 13, 개방 16개)에서 WebAssembly로 20 ms라고 가정한 시간은 노트북의 JavaScript에서 $50 \times 20 = 1000$ ms, 즉 1초가 되고, 더 느린 휴대폰에서는 4초가 됩

니다. 이 수치는 가정입니다. 정책을 정하기 전에 우리 클라이언트의 기기를 직접 측정합니다. 합리적인 정책은 이렇습니다. 페이지가 `fallback` 메시지를 전달하게 하고, 서버에서 이를 지표로 기록하며, 모든 사람이 아니라 WebAssembly를 막는다고 알려진 클라이언트와 일치하는 규칙에 대해서만 난이도를 낮춥니다.

제VIII부: 평가

8.2. Gate의 `admitted` 행에서 $11.84 \times 211,345 / 10^{6} \approx 2.50$이므로 코어 네 개 가운데 약 2.50개가 사용 중이었습니다. 따라서 데몬은 CPU가 병목이 아니었고, 한계를 정한 것은 부하 생성기였습니다. 요청당 CPU 11.84 us는 마이크로초 미만인 분류 비용보다 훨씬 큽니다. 제품 행에는 커널의 소켓 읽기와 쓰기, 파서, 세션 태그 확인, 지표 증가, 응답 포맷, 쓰기 플러시, 그리고 부하 생성기와 연결 스레드 사이의 스케줄러 전환이 모두 들어 있기 때문입니다. 기본 연산 측정 모음은 이 가운데 어느 것도 시간을 재지 않습니다.

제IX부: 운영

9.1. 데이터가 별도의 영수증 갱신보다 먼저 커밋되는 경우, 둘 사이에서 크래시가 나면 옛 영수증이 남아 재시도가 효과를 다시 적용할 수 있습니다. 영수증이 먼저 커밋되는 경우, 데이터를 쓰기 전에 크래시가 나면 재시도가 존재한 적 없는 데이터를 건너뛸 수 있습니다. 원자적 트랜잭션 하나가 두 틈을 모두 없앱니다. 트랜잭션의 확인 응답을 잃는 경우에는 여전히 보관된 재시도가 필요합니다.

9.2. `ip_rules: {"10.20.0.0/16": "ALLOW"}`(또는 점수 100인 `ip_reputation` 행)는 스테이징을 허용합니다. 규칙 `{"name": "admin", "path": "/admin/*", "action": "CHALLENGE", "challenge": {"difficulty": 20, "algorithm": "posw"}}`는 작업을 요구합니다. `ALLOW`와 `"limits": {"rate": 30, "window_seconds": 10}`를 가진, `headers: {"X-Partner-Key": ".*"}`와 일치하는 규칙은 파트너를 10초당 요청 30개로 허용합니다. 규칙의 할당량은 그 규칙과 일치하는 모든 요청에 대해 클라이언트별 데몬 제한(`--rate-limit`, `--rate-window`) 뒤에 확인하므로, 데몬 제한은 적어도 그만큼 넉넉해야 합니다.

제X부: 참고 자료

10.1. 챌린지 레코드에는 그것을 가져온 주소의 키 기반 지문이 담겨 있습니다. 풀이는 새 주소에서 도착하므로 `verify_and_mint`는 증명을 보기도 전에 `FingerprintMismatch`를 반환합니다. 가장 작은 복구 방법은 중간 페이지에 있습니다. 제목이 `CLIENT FINGERPRINT MISMATCH`인 `400`을 받으면 오류를 보여 주는 대신 새 챌린지를 가져와 다시 풉니다. 이미 한 작업은 옮길 수 없기 때문입니다.

조사 이어 가기

버려도 되는 테스트에서 불변식을 한 번에 하나씩 바꿔 봅니다. 스냅숏 재확인을 없애거나, 인시던트 영수증을 빼거나, 속도 간격을 내림해 봅니다. 테스트를 실행하기 전에 실패가 어떤 순서로 일어날지 예측합니다. 쓸모 있는 회귀 테스트는 정상 경로의 출력을 확인하는 데 그치지 않습니다. 빠뜨린 조건이 왜 필요한지 보여 줍니다.

용어집

요청 허용
: 요청을 원본 서버로 전달하기로 하는 결정입니다. Sibuna에서는 유효한 세션, `ALLOW` 규칙, 정적 우회 경로, 평판 `allow` 접두사가 이 결정을 내리며, 어떤 경우에도 WAF의 거부 판정을 뒤집지는 않습니다.

Aho-Corasick 오토마톤
: 고정된 패턴 집합에 속한 패턴을 입력을 한 번 훑으며 찾아내는 유한 오토마톤입니다. Sibuna는 실패 링크를 밀집 전이 테이블에 미리 반영해 두므로 바이트마다 테이블을 한 번만 읽습니다.

차단 테이블
: 차단된 클라이언트 주소와 만료 시각을 담는 락 프리 슬롯 4,096개입니다. 허니팟과 클러스터 전체가 공유하는 평판이 기록하고, 모든 요청이 읽습니다.

캠페인
: 페이로드 임베딩이 서로 코사인 거리 0.35 이내에 있는 보안 인시던트의 묶음입니다. 휴리스틱에 따른 묶음이며, 출처가 같다는 증명이 아닙니다.

챌린지 레코드
: 클라이언트가 풀어야 하는 70자 길이의 상태 없는 식별자입니다. 버전, 알고리즘, 난이도, 개방 수, 발급 시각, 클라이언트 지문, PRF nonce, 규칙 해시, 16바이트 키 기반 BLAKE3 태그를 담습니다. 챌린지를 발급해도 서버에는 아무것도 기록되지 않습니다.

난이도, 작업 비트
: 두 계층의 작업량을 함께 정하는 정수 하나입니다. Hashcash에서는 선행 0비트의 개수이고, 순차 작업에서는 트리 깊이에 3을 더한 값입니다.

Edge
: `--data-dir`(노드 하나의 SQLite 파일) 또는 `--database-url`과 `--cluster-node`(클러스터가 공유하는 PostgreSQL 데이터베이스 하나)를 사용하는 Sibuna 배포입니다. 어느 운영 구성에든 영속 정책, 공유 평판, 인시던트 포렌식을 제공합니다.

엔진 슬롯
: 게시된 정책 엔진이 참조 카운트 포인터로 놓이는 자리입니다. 워커는 분류하는 동안 참조를 쥐고 있습니다. 저장소 스레드는 요청 경로 밖에서 새 엔진을 만들어 포인터 교환으로 게시하고, 이전 엔진은 마지막 읽기 측이 참조를 놓을 때 해제됩니다.

지문
: 클라이언트 주소와 User-Agent의 키 기반 해시입니다. 챌린지와 토큰에 묶여 있으므로 어느 쪽도 다른 클라이언트에서 재전송할 수 없습니다.

forward-auth
: 인그레스가 Sibuna에 요청을 허용할지(`200`), 아니면 챌린지를 제시하거나 거부하거나 제한할지(`401`, `403`, `429`) 묻고, 원본 서버로의 프록시는 인그레스가 직접 맡는 배포 모드입니다.

Gate
: 작업 증명 기반 요청 허용, 세션, 규칙, 평판, GCRA 제한과 차단을 갖추었지만 애플리케이션 검사는 하지 않는 운영 구성입니다(`--gate`).

GCRA
Generic Cell Rate Algorithm입니다. 클라이언트마다 이론적 도착 시각 하나만 두고, N개의 버스트를 허용한 뒤 방출 간격마다 요청 하나를 허용합니다. 윈도 경계가 없습니다.

Hashcash
첫 번째 계층입니다. 챌린지, 콜론, 십진수 nonce를 이어 붙인 값의 SHA-256이 선행 0비트 b개를 갖도록 하는 nonce를 찾습니다. 기대 시행 횟수는 2^b이며, 해시 한 번으로 검증합니다.

허니팟
`/__sibuna/honeypot`으로 향하는 보이지 않는 링크입니다. 이 링크를 따라간 클라이언트는 차단되고, 평판 −100인 인시던트가 기록됩니다.

유휴 연결 회수기
`--idle-timeout`보다 오래 아무것도 주고받지 않은 연결을 닫는 스레드입니다. 느린 클라이언트가 연결 스레드를 붙잡아 둘 수 없게 합니다.

중간 페이지
보호된 페이지 대신 제공하는 내장 HTML 페이지입니다. 챌린지를 가져와 Web Worker에서 풀고, 풀이를 보낸 뒤 페이지를 다시 불러옵니다.

JA4
FoxIO가 정의한, 클라이언트가 TLS 연결을 여는 방식에 대한 36문자 지문입니다. 프로토콜, TLS 버전, 서버 이름, 암호 스위트와 확장의 개수, ALPN, 그리고 정렬한 암호 스위트와 확장의 해시를 담습니다. Sibuna는 이 지문을 계산하지 않습니다. 신뢰할 수 있는 인그레스가 `--ja4-header`로 지정한 헤더에 담아 전달하는 값을 읽고, 규칙은 `ja4` 필드로 이 값을 비교합니다(SID 0014). 챌린지와 토큰에 묶이는 클라이언트 지문과는 관계가 없습니다.

키 기반 BLAKE3
Sibuna의 키 스케줄, 챌린지 태그, 세션 태그, 지문, 챌린지 nonce의 바탕이 되는 의사 난수 함수입니다.

MAC 토큰
기본 세션 쿠키입니다. 40바이트 페이로드(버전, 작업 수준, 타임스탬프, 만료, 규칙 해시, 지문)와 16바이트 키 기반 BLAKE3 태그로 이루어지며, 상수 시간에 검증합니다.

작업 수준
세션 보유자가 실제로 푼 방식과 작업 비트입니다. 토큰에 담기며, 경로는 세션의 수준이 그 경로의 요구 수준에 이를 때에만 그 세션을 허용합니다.

개방
순차 작업 증명에서 잎 라벨 하나와, 그 잎에서 루트까지의 경로에 있는 형제 라벨입니다. 검증자는 경로를 다시 계산해 커밋먼트와 비교합니다.

순차 작업 증명(PoSW)
두 번째 계층입니다. 라벨이 앞선 모든 라벨에 의존하는 Cohen-Pietrzak 해시 그래프로, 증명자가 코어를 몇 개 가지든 순차 해시 $2^{n+1}-1$번을 강제합니다. 검증에는 해시 $t(n+1)$번이 듭니다.

평판 트라이
정책 파일과 `ip_reputation` 테이블에서 온 `allow`, `deny`, `challenge` 접두사를 담는 128비트 기수 트라이입니다.

Robin Hood 사용 완료 집합
해결된 챌린지 태그를 담는 고정 용량의 개방 주소법 테이블입니다. 유효한 풀이를 두 번 제출하지 못하게 합니다.

규칙 해시
챌린지를 요구한 규칙 이름의 64비트 해시입니다. 챌린지, 토큰, `X-Sibuna-Rule-Hash` 헤더에 담깁니다.

Shield
기본 운영 구성입니다. Gate에 의미 기반 방화벽을 더합니다(`--shield`).

SID
Sibuna Discussion 기록입니다. `docs/sid/records` 아래에 있는 논문 형식의 설계 문서로, 시스템의 한 부분을 뒷받침하는 가정, 증명, 측정을 담습니다.

운영 구성
두 가지 제품 형태인 Gate와 Shield 중 하나입니다. 저장소와 클러스터는 어느 쪽에서든 선택할 수 있습니다.

WEIGH
즉시 결정하는 대신 요청 점수에 부호 있는 가중치를 더하는 규칙 동작입니다. 합계가 임계값 이상이면 추가 작업 비트로 챌린지를 제시하거나 거부합니다.

Zaxonlite
SID 0005의 내장 SQLite 복제 엔진(Multi-Paxos)으로, SID 0011에서 SQLite와 PostgreSQL 저장소로 대체되었습니다. 지금은 그 클러스터 옵션을 지정하면 `RemovedOption` 오류로 시작이 중단됩니다.

참고 문헌

- Aho, A. V., and Corasick, M. J. “Efficient string matching: an aid to bibliographic search.” *Communications of the ACM* 18(6), 1975.
- ATM Forum. *Traffic Management Specification Version 4.0*, af-tm-0056.000, 1996 (Generic Cell Rate Algorithm).
- Back, A. "Hashcash: a denial of service counter-measure." 기술 보고서, 2002.
- Blocki, J., Lee, S., and Zhou, S. “On the security of proofs of sequential work in a post-quantum world.” *Information-Theoretic Cryptography (ITC)*, 2021.
- Celis, P. *Robin Hood Hashing*. 박사 학위 논문, University of Waterloo, 1986.
- Cohen, B., and Pietrzak, K. “Simple proofs of sequential work.” *EUROCRYPT*, 2018.
- Dwork, C., and Naor, M. “Pricing via processing or combatting junk mail.” *CRYPTO*, 1992.
- Fiat, A., and Shamir, A. “How to prove yourself: practical solutions to identification and signature problems.” *CRYPTO*, 1986.
- Fielding, R., Nottingham, M., and Reschke, J. *HTTP Semantics*, RFC 9110 및 *HTTP/1.1*, RFC 9112. IETF, 2022.
- FoxIO. *JA4+ Network Fingerprinting*. https://github.com/FoxIO-LLC/ja4, 2023 (JA4 TLS 클라이언트 지문).
- Hanson, N. “libinjection: SQL injection detection by tokenization.” Black Hat USA, 2012.
- Juels, A., and Brainard, J. “Client puzzles: a cryptographic countermeasure against connection depletion attacks.” *NDSS*, 1999.
- Lamport, L. "The part-time parliament." *ACM Transactions on Computer Systems* 16(2), 1998 및 "Paxos made simple." *ACM SIGACT News* 32(4), 2001.
- Mahmoody, M., Moran, T., and Vadhan, S. “Publicly verifiable proofs of sequential work.” *Innovations in Theoretical Computer Science (ITCS)*, 2013.
- O'Connor, J., Aumasson, J.-P., Neves, S., and Wilcox-O'Hearn, Z. *BLAKE3: one function, fast everywhere*. 명세, 2020.
- Vyukov, D. "Bounded MPMC queue." 1024cores.net, 2011 (인시던트 큐에 쓰는, 시퀀스 번호를 붙인 링).
- Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., and Zhu, H. “Hyperscan: a fast multi-pattern regex matcher for modern CPUs.” *NSDI*, 2019.
- Sibuna Discussions 0001부터 0014까지, `docs/sid/records`, 2026: 프로세스, 기반 아키텍처, 선언형 정책 엔진, 의미 기반 검사, Zaxonlite 저장소 아키텍처, 수학적 기초, 콘솔 관리 인터페이스, AI 봇 트래픽 모니터링, 청크 요청 본문, Rust 포팅, sqlx를 통한 저장소, rustls를 통한 콘솔의 아웃바운드 TLS, 콘솔 저장소, 인그레스에서 받는 신뢰할 수 있는 JA4 지문.

색인

A
Aho-Corasick 오토마톤, **139**

E
Edge, **139**

F
forward-auth, **139**

G
Gate, **139**
GCRA, **140**

H
Hashcash, **140**

J
JA4, **140**

M
MAC 토큰, **140**

R
Robin Hood 사용 완료 집합, **140**

S
Shield, **141**
SID, **141**

W
WEIGH, **141**

Z
Zaxonlite, **141**

ㄱ
개방, **140**
규칙 해시, **141**

ㄴ
난이도, 작업 비트, **139**

ㅅ
순차 작업 증명(PoSW), **140**

ㅇ
엔진 슬롯, **139**
요청 허용, **139**
운영 구성, **141**
유휴 연결 회수기, **140**

ㅈ
작업 수준, **140**
중간 페이지, **140**
지문, **139**

ㅊ
차단 테이블, **139**
챌린지 레코드, **139**

ㅋ
캠페인, **139**
키 기반 BLAKE3, **140**

ㅍ
평판 트라이, **140**

ㅎ
허니팟, **140**