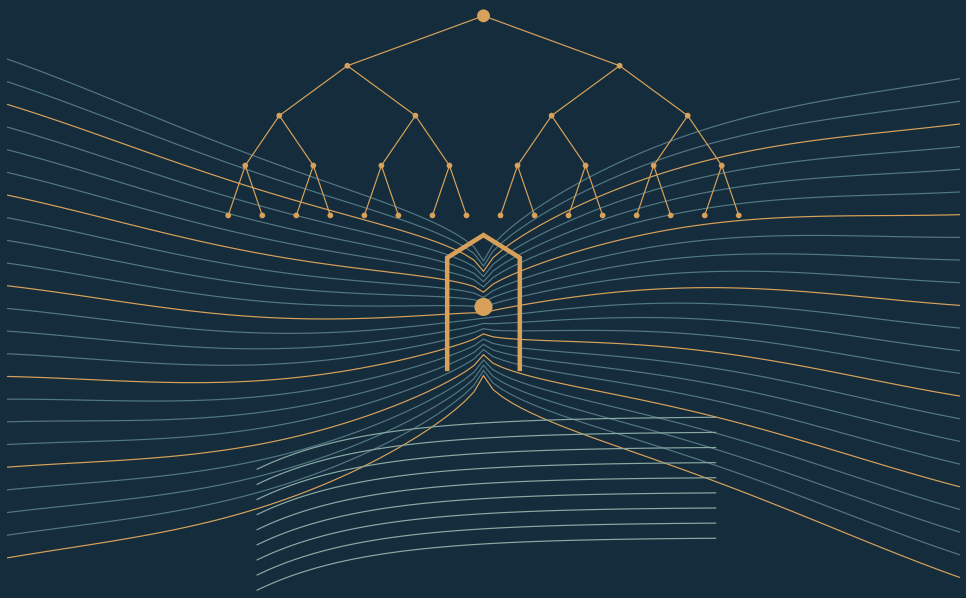
技術ガイド

SIBUNA

Web 防御の数学とエンジニアリング



作業 · 信頼 · 有界な状態

Vikrant Rathore（協力：Ronak Rathore、Kanak Rathore）

バージョン 1.0.0-beta.3 · 11 月 13, 2026

目次

序文 **6**

本書の読み方 **8**

三つの読み方 8

本書の約束事 9

第 I 部：リクエストのコスト **10**

許可は経済的な判断 10

作業証明が示すもの 10

確率変数としてのハッシュ探索 11

セッションによる作業の償却 12

有界な状態という第二の予算 12

本書で保ち続ける不変条件 12

第 II 部：先行事例と設計空間 **14**

Sibuna の由来 14

一つのバイナリに三つの考え方 16

機能比較 17

第 III 部：暗号研究からの基礎 **23**

Hashcash：第一層の数学 23

逐次作業証明：第二層 24

対称鍵による認証：チャレンジとセッション 27

第 IV 部：Sibuna プロトコル **31**

チャレンジの往復 31

状態を持たないチャレンジレコード 33

検証の順序と一回限りの使用 34

セッション Cookie 35

第 V 部：Rust によるゼロアロケーションのエンジニアリング **38**

接続のループ 38

割り当てを行わない並行状態 49

第 VI 部：オートマトン、トライ、意味解析ファイアウォール **59**

一回の走査による複数パターンの照合 59

IPv4 と IPv6 の基数トライ 61

意味解析ファイアウォール 61

宣言的ポリシーエンジン 63

第 VII 部：ブラウザーの作業証明エンジン **65**

一つのソース、二つのターゲット 65

ワーカーのプロトコル 69

較正 70

中間ページ 70

第 VIII 部：実験による評価 **72**
方法 72
プリミティブのレイテンシー 76
製品全体の比較 80
同じ負荷を与えたときのレイテンシー 82
Sibuna と Anubis 84
コンソールのオーバーヘッド 88
測定していないもの 89
第 IX 部：運用とデプロイ **92**
運用構成と設定 92
コンソールのプレビュー 95
デプロイの構成 113
宣言的なポリシー 121
永続ストレージとクラスタリング 121
パッケージング 126
第 X 部：手元リファレンスと診断 **131**
エンドポイント 131
メトリクス 132
ステータスコード 133
エラーの一覧 134
ポリシーのスキーマ 136
ストレージのテーブル 137
ビルドターゲット 137
第 XI 部：早見表 **139**
ビルド 139
実行 139
コンソール（オプトインのプレビュー） 139
まず押さえるべきフラグ 140
/__sibuna/配下のエンドポイント 141
ステータスコード 141
ヘッダー 142
ポリシーファイルの骨組み 142
SQL で見るストレージ（Edge） 142
診断 143
覚えておく数値 143
イングレスのルーティング 143
解答例（抜粋） **145**
第 I 部：コストと確率 145
第 II 部：先行事例 145
第 III 部：暗号 145
第 IV 部：プロトコル 146
第 V 部：状態と順序 146

第 VI 部：アルゴリズム . . . 146
第 VII 部：ブラウザーエンジン . . . 147
第 VIII 部：評価 . . . 147
第 IX 部：運用 . . . 147
第 X 部：リファレンス . . . 147
さらに調べるために . . . 148
用語集 . . . **149**
参考文献 . . . **152**
索引 . . . **153**

© 2026, Vikrant Rathore および Sibuna のコントリビューター

Web 防御の数学とエンジニアリング。

序文

リクエストは、メソッド、パス、いくつかのヘッダー、そしてサーバーの処理を求める要求を携えて届きます。これらのバイトのどれも、そのリクエストに応える価値があることを示しません。Sibuna はリクエストとオリジンの間に一つの判断を置きます。許可するか、拒否するか、送信者に検証可能な作業を求めるかです。本書はこの判断を、数学的なモデルからメモリ上のバイトまで追います。

このシステムには二つの運用構成があります。*Gate* は作業証明による許可と、ローカルの大量アクセス対策を提供します。*Shield* はアプリケーションの検査とポリシーによる判断を加えます。どちらも、ポリシー、レピュテーション、インシデント履歴のために永続ストレージを使えます。ストレージは一つのノードに置くことも、クラスターで共有することもできます。このデプロイを *Edge* と呼びます。有効なセッションは許可を確立しますが、リクエストを検査から免除するわけではありません。永続的な状態をクラスターで共有すれば状態は分散されますが、ローカルのレートカウンターがクラスター全体のクォータに変わるわけではありません。

工学上の中心的な問いは、ハッシュが速いかどうかではありません。信頼できないクライアントが入力を選ぶときに、処理経路全体が有界であり続けるかどうかです。パーサーは何バイトまで検査できるでしょうか。チャレンジはどれだけのメモリを消費し得るでしょうか。クライアントはいくつの接続を開いたままにできるでしょうか。どの書き込みなら再試行してよいでしょうか。ノードやデータベースが消えた後に、何が利用できるでしょうか。どの答えも、不変条件とその境界の両方を示さなければなりません。

数学も同じ目的に役立ちます。期待コストはレイテンシーの保証ではありません。固定したコミットメントに対するサンプリングの議論は、あらゆる適応的な証明者に対する証明ではありません。キューはバーストを吸収しますが、恒常的に遅い消費側を補うことはできません。本書では小さなモデルを導き、例題を手で解き、そのうえで実装がモデルからどこで離れるかを問います。

本書は三通りに使えるように書いています。順に読めば、第 I 部から第 VII 部は一続きの講座になっています。各章は、読み終えたときにできるようになることを最初に示し、例題を手で解き、演習と、自分の言葉で説明する不変条件で締めくくります。抜粋した解答例を巻末に載せています。第 VIII 部から第 X 部はリファレンスです。出所を明記した測定、すべてのフラグと構成、すべてのエンドポイント、ステータス、エラー、テーブルを扱います。第 XI 部はターミナルのそばに置く 2 ページの早見表です。最後に用語集と参考文献があります。

第 II 部では、Sibuna を Anubis、SafeLine、Cloudflare WAF と機能ごとに比較します。第 VIII 部では、単一のプリミティブから製品全体の実行まで Sibuna 自身の測定を示し、同じ負荷生成器の下で製品全体として Sibuna と Anubis を比較します。どちらも読者が主張を確かめられるように書いています。比較はその出典を示し、どの測定も、値を写したコミット済みの結果ファイルを示します。

読者には、プログラミングの基礎、対数、条件付き確率の知識を想定しています。Rust は、言語の概観ではなく、所有権とデータ配置を通して紹介します。各章は順に読めます。コスト

モデルがプロトコルの動機となり、プロトコルが状態を決め、状態が並行処理とストレージの設計を決めます。後の章では、これらの不変条件を測定と運用手順に落とし込みます。

本書は実装の本です。計算パズルが自動化されたトラフィックをなくすと主張するものではありません。クライアントは計算資源を買えますし、アドレスは共有され得ますし、アプリケーションの構文は上限のある検出器よりも豊かです。本書が目指すのは、コストと限界をスローガンの陰に隠さず、調べ、テストし、変更できるシステムです。

***Sibuna 1.0.0-beta.3* 対応版、*2026* 年 *11* 月**

ソース、演習、再現可能な測定は Sibuna リポジトリにあります。

本書の読み方

一つの問いから始めます。サーバーが今下せる、安全で最も安い判断は何でしょうか。プロトコルと実装の章を通して、この問いを手元に置いておきます。誤ったリクエストを許可する安価な処理は欠陥です。コストに上限のない正しい処理も、別の種類の欠陥です。

問い	答えを扱う箇所
パズルで何が得られるか	第 I 部から第 III 部：コスト、確率、証明の限界。
Sibuna は他とどう違うか	第 II 部：系譜と機能表。第 VIII 部：測定。
セッションは何を意味するか	第 IV 部：結び付け、検証の順序、期限、リプレイ。
メモリはどこに使われるか	第 V 部と第 VI 部：バッファー、スレッド、テーブル、オートマトン、公開。
ブラウザーはどう解くか	第 VII 部：ワーカーのメッセージ、メモリ、較正。
動作をどう確かめるか	第 VIII 部：三つのハーネスと、それぞれが対象外とするもの。
どう運用するか	第 IX 部：フラグ、構成、ポリシー、ストレージ、クラスター、パッケージ。
このエラーは何か	第 X 部：エンドポイント、ステータス、エラー、スキーマ。第 XI 部：早見表。

三つの読み方

- 設計を学ぶ。第 I 部から第 IV 部を順に読み、次に第 VI 部の実行トレースの例、そして第 VII 部に進みます。演習を解いてください。巻末の解答と比べるのはその後にします。トレースが共有テーブルや借用したスライスに達したら、第 V 部に戻ります。
- デプロイを運用する。第 IX 部、次に第 XI 部を読み、参照用に第 X 部を開いておきます。運用構成を選ぶ前に第 II 部の「機能比較」を、ワーカー数と接続数の上限を選ぶ前に第 VIII 部の「製品全体の比較」を読んでください。
- コードを変更する。第 V 部と第 VI 部、各章のソース参照、そこで引用している SID の記録を読みます。第 VIII 部では、変更した部分のリグレッションをどのハーネスが捉えるかを説明します。第 II 部には、変更後も守るべき制約をまとめています。

本書の約束事

例題は答えだけでなく途中の状態も示します。演習は仮定を一つ変えるよう求めます。ヒントは最初の一歩を示し、巻末に抜粋した解答例で推論を確かめられます。図はリクエストの経路とバックグラウンドの処理を区別します。ソース参照は、本文とコードが食い違って見えるときに調べるべき実装を示します。「実装」という見出しの囲みは関数とそのファイルを示します。「不変条件を説明してください」という見出しの囲みでは、その考え方を自分の言葉で説明し直します。

式では、bはビット単位の Hashcash の難易度、nは逐次作業の深さ、tは開示の数、pは試行ごとの成功確率、Kは試行回数、Tはレートリミッターの送出間隔、τはそのバースト許容量、Qはキューの容量です。特に断らない限り、記号はその節の中だけで使います。第 VIII 部のナノ秒や毎秒リクエスト数は、名前を示した実行での測定値です。例題の数値は選んだ入力であり、ベンチマークの主張ではありません。統計モデルは、結論を導く前に仮定を述べます。

第 I 部

リクエストのコスト

アルゴリズムを選ぶ前に、それが守るべき資源を特定します。この章では、後のすべての設計判断を測る基準となる小さなモデルを作ります。オリジンの作業、パズルの作業、セッションによる償却、そして信頼できないクライアントがサーバーに保持させ得るメモリです。

この章の学習目標 この章を読み終えると、ゲートが元を取るかどうかを決める不等式を書き、作業証明が何を示し何を示さないかを述べ、ハッシュ探索の期待値と裾を導き、セッションが作業を償却する仕組みを説明し、本書の残りで保つ五つの不変条件を挙げられるようになります。

許可は経済的な判断

オリジンが、許可したリクエストごとにデータベースを検索し、ページを描画するとします。その作業をc_o、リクエストあたりのゲートの作業をc_g、到着率をrとします。ゲートがなければ、オリジンは毎秒rc_o単位の作業をこなさなければなりません。割合aだけを許可するゲートは、この需要をrac_oに変え、自身はrc_gを費やします。ゲートが役に立つのは、節約したオリジンの作業が、ゲート自身のコストと正当な訪問者に課すコストを上回るときだけです。

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

これは処理能力のモデルであり、測定ではありません。帯域幅、接続の状態、ストレージ、クライアントの待ち時間を省いています。このモデルの価値は、欠けている項がすべて目に見えることです。キャッシュにヒットすればc_oは小さくなり、高価なクエリなら大きくなります。要求側とサーバー側のコスト比に普遍的な値はありません。

例題：ゲートが元を取るとき $c_o = 100$作業単位、$c_g = 1$とします。リクエストの半分を拒否すれば、ゲート付きのシステムは到着一件あたり 100 単位ではなく$1 + 0.5 \times 100 = 51$単位を費やします。千件に一件しか拒否しなければ100.9単位を費やします。このモデルでは、ゲートは節約する以上のコストがかかります。ここで選んだ数値は時間の測定値ではありません。性能に関するどの主張にもワークロードを添えるべき理由を示しています。

作業証明が示すもの

クライアントパズルが示すのは、誰かが公開された検証規則を満たす入力を見つけたということです。人間であること、身元、善意は示しません。要求側は計算資源を借りることも、有効期間内の有効なセッションを再利用することも、作業を複数のマシンに分散することもで

きます。パズルは許可のコストを変えます。許可されたリクエストに何を許すかは、引き続きポリシーと検査が決めます。
アドレスに基づく制限はこれを補います。制限は一つのアドレスを縛りますが、共有アドレスの背後には多くの人がいることがあり、一人の要求者が多くのアドレスを使うこともあります。パズルもアドレスも、身元を判定するオラクルにはなりません。そのため Sibuna は、許可、検査、レピュテーションを別々の判断として扱います。

確率変数としてのハッシュ探索

理想的な 256 ビットのダイジェストでは、先頭にb個のゼロビットを要求すると、独立な試行ごとの成功確率は$p = 2^{-b}$になります。最初の成功までの試行回数をKとします。このとき

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

期待値は裾の和から導かれます。

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

難易度を 1 段上げると、期待試行回数は 2 倍になります。どのパズルも 2 倍の時間がかかると約束するものではありません。最初の試行で成功するものもあれば、平均よりはるかに長くかかるものもあります。解答を一度だけ測ったクライアントが測ったのは、主にこのランダム性です。



図 1 ハッシュ探索の生存確率。期待試行回数のx倍だけ試行した後も探索がまだ続いている確率で、作業量が大きいときの近似$P(K > x/p) \approx e^{-x}$を使います。

例題：期待値は締め切りではない $b = 16$では、期待作業量は 65,536 回のハッシュです。それだけ試行した後もまだ探索を続けている確率は、およそ$e^{-1} = 0.368$です。試行回数の 95 パーセンタイルは$\lceil \ln(0.05)/\ln(1-2^{-16}) \rceil \approx 196327$で、平均のほぼ 3 倍です。ユーザーインターフェースは、期待完了時刻に失敗を告げるのではなく、このばらつきを許容すべきです。

セッションによる作業の償却

セッションが期限までにm件のリクエストを許し、パズルのコストがc_p、検証のコストがc_v、セッションの確認のコストがc_sだとします。失敗した試みを無視すると、リクエストあたりの償却したゲートのコストは$c_v/m + c_s$、クライアントのパズルのコストはc_p/mです。セッションの有効期間を延ばすと、人にも自動化されたクライアントにも同じように有利になります。有効期間の選択はポリシーの判断であり、暗号上の最適化ではありません。

演習 1.1. クライアントはセッションごとに 100 件のリクエストを送り、100 万回の試行を要するパズルを解きます。リクエストあたりの償却した作業量はいくつでしょうか。そのセッションを二つ目のプロセスと共有すると、何が変わるでしょうか。
ヒント：会計上のモデルと、トークンが実際に結び付けるものとを区別してください。

有界な状態という第二の予算

作業をリクエストの経路から移しても、作業は消えません。インシデントが到着率λで届き、ストレージスレッドが平均速度μで永続化し、キューがQ件のレコードを保持するとします。$\lambda > \mu$のとき、流体近似から次が得られます。

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

大きなキューは時間を稼ぎますが、スループットを生み出しはしません。バッチ処理はトランザクションのコストを償却しますが、保持したレコードには依然としてメモリと、いずれ行う処理が必要です。過負荷のとき、Sibuna は新しいインシデントのレコードを破棄し、その件数を数えます。リクエストの判断はメモリ上の状態から引き続き行います。運用者は、HTTP の成功率だけでなく、失われた証跡も監視しなければなりません。

演習 1.2. キューは 512 件のレコードを保持します。到着率は 1,000/s、排出率は 600/s です。最初に空のキューは、超過分をどれだけの時間吸収できるでしょうか。この答えが近似にすぎないのはなぜでしょうか。
ヒント：速度の差を使い、次にバーストとバッチコミットを考えてください。

本書で保ち続ける不変条件

1. 信頼できないリクエストは、上限のないサーバー状態を作れません。
2. 有効な許可トークンがあっても、アプリケーションによる拒否は回避されません。
3. バッファーは、そこから借用したどのスライスよりも長く生存します。
4. 永続的な再試行によって、インシデントやそのレピュテーションへの効果が重複することはありません。
5. 公開したポリシーのスナップショットは、最後の読み手が手放すまで変更されません。

本書の残りでは、これらの主張を真にする仕組みと、違反を明らかにするテストを導きます。高速パスが役に立つのは、これらの主張が真であり続ける間だけです。

不変条件を説明してください. ゲートの不等式$c_g < (1-a)c_o$を言葉で述べてください。次に、測定した検証のレイテンシーだけでは、ゲートを運用する価値があるかどうかを運用者が判断できない理由を説明してください。

第 I I 部

先行事例と設計空間

Sibuna が学んだシステムの中に *Sibuna* を位置付けます。暗号の文献に由来するクライアントパズルと逐次作業、シグネチャ型と意味解析型のアプリケーションファイアウォール、作業証明の中間ページ、ホスト型のエッジネットワークです。次に、運用者が比較検討する可能性の最も高い四つのツールを機能ごとに比べ、どの事実を確認したかを記録します。

Sibuna の由来

この章の学習目標　この章を読み終えると、1992 年から 2018 年の逐次作業の構成に至るクライアントパズルの系譜をたどり、Sibuna が組み合わせる三つのアーキテクチャ上の考え方を挙げ、Sibuna、Anubis、SafeLine、Cloudflare WAF を一つの設計空間に位置付け、比較の主張のうちどれを検証し、どれを文書から読んだだけかを述べられるようになります。

クライアントパズルと逐次作業

Dwork と Naor の「Pricing via Processing」（CRYPTO 1992）は、サービスが要求者のために作業をする前に、ほどほどに難しく容易に確認できる計算を求めることを提案しました。Back の Hashcash（1997 年、2002 年に論文化）は、パズルを難易度が一つの整数で決まるハッシュの部分逆算にしました。Juels と Brainard のクライアントパズル（NDSS 1999）は、この考え方を接続の大量アクセスに適用しました。そこではサーバーは未解決のパズルについて何も記憶してはなりません。三つはいずれも、Sibuna が受け継ぐ形を共有しています。サーバーが選ぶステートメント、コストを調整できる解答、そして解くより桁違いに安い検証です。ハッシュ探索は自明に並列化できるので、コアの多い要求者ほど早く終わります。Mahmoody、Moran、Vadhan（CRYPTO 2013）は逐次作業の証明を定義しました。そこでは証明者は、コア数にかかわらず、依存するステップの長い連鎖を実行しなければなりません。Cohen と Pietrzak（EUROCRYPT 2018）は、検証者に$O(t \log N)$の作業しか必要としない単純なハッシュグラフの構成を示しました。Blocki、Lee、Zhou（ITC 2021）は、同じ構成が量子ランダムオラクルモデルにおいて量子の証明者に対しても健全であることを証明しました。第 III 部では、この研究の流れから Sibuna の第二層を導きます。前提は SID 0006 に記録しています。

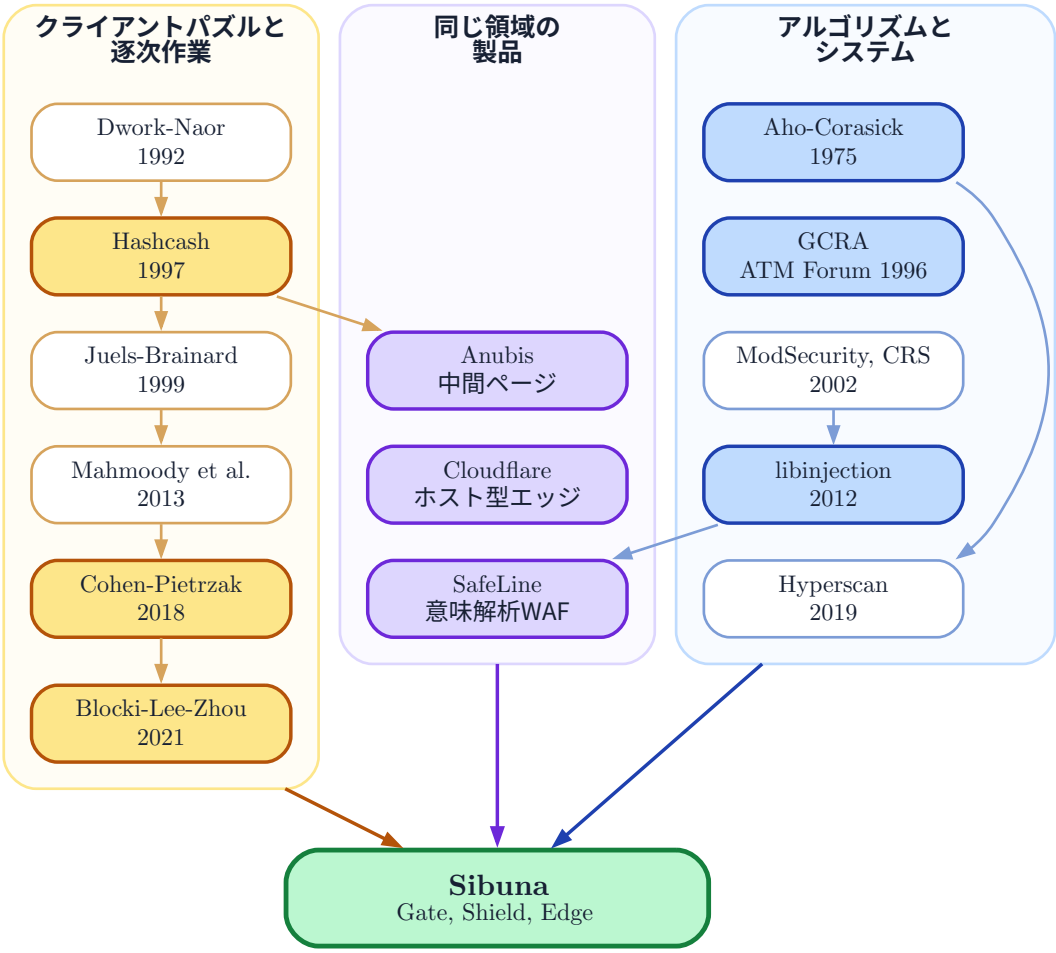


図 2 Sibuna が組み合わせる考え方の系譜。左の列はクライアントパズルと逐次作業の暗号、右の列は Sibuna が土台とするアルゴリズムとシステム、中央の列は同じ領域の製品で、時間は上から下へ進みます。細い矢印は直接の継承を表し、塗りつぶした箱は Sibuna が直接取り入れた考え方です。以前のストレージ設計は Multi-Paxos（Lamport 1998、SID 0005）に基づいていました。現在の Sibuna は状態を SQLite または PostgreSQL に保存します（SID 0011）。

アプリケーションファイアウォール

第一世代の Web アプリケーションファイアウォールは、リクエストのバイトを正規表現と照合していました。ModSecurity（2002）と OWASP Core Rule Set はこの方式を公開し広く普及させ、同時にそのコストも周知のものにしました。リクエストごとに数百のバックトラックするパターン、手で調整する異常スコア、普通の入力での誤検知の長い裾です。libinjection（Hanson、2012）は、パターンのリストを、文字列が SQL としてパースできるかを問うトークナイザーに置き換え、コストと誤検知の両方を減らしました。Chaitin の SafeLine はこの意味解析の考え方を完全な製品に持ち込み、Hyperscan（Wang ら、NSDI 2019）は入力が長い場合にベクトル化したリテラル照合をどこまで推し進められるかを示しました。第 VI 部で

は、Sibuna が検査する短いフィールドに対して、正規表現でも SIMD エンジンでもなく、密なオートマトンと小さな構造トークナイザーを選んだ理由を説明します。

中間ページとエッジ

Web サイトの前に置く作業証明の中間ページは 2025 年に一般的になりました。言語モデルの学習に供するスクレイパーのトラフィックによって、普通のオリジンの運用費用がまかなえなくなったからです。Anubis（Techaro）は最もよく知られたオープンな実装です。JavaScript のパズルを配信し、セッショントークンに署名し、それ以外はトラフィックを通過させる Go のリバースプロキシです。Sibuna の Gate 運用構成は同じ問題を、状態を持たないチャレンジ、対称鍵のトークン、第二の作業の層で解きます。両者の違いは以下の比較に記録しています。規模の反対の端では、Cloudflare のようなホスト型のエッジが、検査、レート制限、ボットのスコア付けを、グローバルなコントロールプレーンを持つ数百の拠点に配置しています。Sibuna の Edge 構成はその形（ローカルで不変のスナップショットを供給する、共有のポリシーとレピュテーションのプレーン）を借りつつ、セルフホストのままです。そのためストレージ層はサービス呼び出しではなく、運用者が運用するデータベース（一つのノード上の SQLite ファイル、またはクラスターで共有する一つの PostgreSQL データベース）で、ストレージスレッドがリクエストの経路の外で読み込みます。

一つのバイナリに三つの考え方

Sibuna は、許可のためのブラウザーでの作業チャレンジ、上限のあるアプリケーションペイロードの検査、ローカルのポリシースナップショットを公開する共有のコントロールプレーンを組み合わせます。Gate と Shield が二つの製品の運用構成で、ストレージと分散デプロイはどちらでも選択できます。

特性	Gate	Shield
許可	ネイティブの Hashcash または逐次作業の検証	同じ
セッション	鍵付き BLAKE3、任意で Ed25519	同じ
ローカルの制御	ルール、CIDR、GCRA、禁止	同じ
検査	無効	上限のあるシグネチャと構造トークナイザー
状態	任意の永続的なポリシーとレピュテーション、クラスターで共有	検査のフォレンジックも

ブラウザーのソルバーは、ネイティブの検証器と同じ証明のソースからコンパイルされます。密なリテラル照合はバックトラックを避け、走査の作業量を入力の長さに比例させます。不変のポリシースナップショットは、リクエストの処理を SQL とデータベースの可用性から切り離します。これらの選択のどれも、アルゴリズムとして普遍的に最適であることを示すものではありません。短いヘッダー、長いボディー、パターンの多様さ、コア数、キャッシュサイズによって、トレードオフは変わります。

設計上の制約として採用したもの

1. リクエストの経路でヒープ割り当てをしません。どの表も固定の容量と、明示した飽和時の振る舞いを持ちます。
2. チャレンジの発行では何も書き込みません。メモリを占めるのは、対価を支払った解答だけです。
3. セッションは許可を与えますが、リクエストを検査や明示的な拒否から免除することはありません。
4. 主張するすべての数値は、コミット済みのハーネスで測定したものか、モデルであると明記したもののいずれかです。
5. 暗号の選択は、発表の場と証明を引用し、量子計算への姿勢を述べます。

機能比較

以下の四つの表で、Sibuna と、よく並べて挙げられる三つのツールを比較します。他の製品に関する事実は、2026 年 9 月に公開ドキュメントとリリース成果物から読み取りました。対象は、Anubis 1.27.0（MIT ライセンス、Go）、SafeLine コミュニティ版 9.x（非公開の検出器イメージを取り巻く GPL-3.0 の管理コード、Docker Compose）、Cloudflare WAF の開発者向けドキュメントです。Anubis はベンチマーク用のホストでも実行しましたが、SafeLine と Cloudflare は実行していません（理由は第 VIII 部で説明します）。ダッシュはその製品がその機能を提供しないことを、プラン名はその機能がそのプランでだけ提供されることを意味します。Sibuna の各測定値には、出典となるコミット済みの結果ファイルを示しています。TLS フィンガープリントの行は 2026 年 10 月に、Anubis の変更履歴（1.21.0 と 1.26.0）、SafeLine のリリースノート（8.10.0 と 9.2.0）、Cloudflare のページ「JA3/JA4 fingerprint」と照らし合わせて確認しました。

表 3 機能比較、その 1：デプロイ

機能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
実行形態	一つのバイナリ。リバースプロキシまたは forward auth。任意で SQLite ファイルまたは共有 PostgreSQL データベース	一つの Go バイナリ。リバースプロキシまたは forward auth	七つの Docker コンテナー（tengine、detector、mgt、luigi、fvm、chaos、PostgreSQL）	ホスト型のネットワーク。DNS を Cloudflare に向けます
言語、ランタイム	Rust、ガベージコレクターなし。ストレージを無効にすると外部クレートなし	Go、ガベージコレクションあり	C/C++のプロキシと検出器、Go の管理部、PostgreSQL	独自のエッジソフトウェア

機能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
ライセンス	LGPL-3.0-or-later	MIT	GPL-3.0 の管理コード。検出器のイメージはクローズド	独自のサービス
ホストでの占有量	35.7 MiB のバイナリ（`benchmarks/results/comparison-latest.json`）。アイドル時の RSS は 5.9 MiB（`benchmarks/results/primitives-latest.json`）。負荷時のピーク RSS は最大 16.0 MiB（`benchmarks/results/product-latest.md`）	38.1 MiB のバイナリ。アイドル時の RSS は 21 MiB。Linux での検証環境、コンソールは無効	最小で CPU コア 1 つ、RAM 1 GB、ディスク 5 GB、Docker 20.10 以降	オンプレミスには不要

表 4 機能比較、その 2：許可とセッション

機能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
作業証明による許可	Hashcash（ビット単位）と Cohen-Pietrzak の逐次作業をルールごとに選択。WebAssembly と JavaScript による代替手段	16 進ニブル単位の SHA-256 Hashcash（`fast`、`slow`）。作業を伴わない `metarefresh` と `preact` のチャレンジ	JavaScript によるボット対策チャレンジと CAPTCHA。作業量の裏付けなし	マネージドチャレンジ、JS チャレンジ、Turnstile。Bot Fight Mode は計算チャレンジを発行
サーバー上のチャレンジの状態	解かれるまでなし。解かれたタグは固定の Robin Hood 表に	ストアのバックエンド：メモリ、bbolt、Valkey、S3	スタックが管理	Cloudflare が管理
セッショントークン	鍵付き BLAKE3 タグ、支払った作業に結び付いた 75 文字の Cookie。任意で Ed25519	Ed25519 で署名した JSON Web Token。任意で HS512	スタックが発行する Cookie	`cf_clearance` Cookie

機能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
標準のトークンの耐量子性	対称鍵：適用されるのは Grover による二次の高速化だけ	Ed25519：Shor のアルゴリズムで破られる	記載なし	記載なし
ポリシー言語	JSON のルール：パス、User-Agent、ヘッダー、IPv4/IPv6 CIDR、JA4 フィンガープリント。ALLOW、DENY、CHALLENGE、しきい値付きの WEIGH。データベースからの動的なルール	YAML/JSON の bots：UA、パス、ヘッダーへの正規表現。CIDR。CEL 式。しきい値付きの WEIGH	コンソールで定義するカスタムルール、ACL、IP グループ	Rules language 式。プランにより *5* / *20* / *100* / *1000* 個のカスタムルール
TLS フィンガープリント（JA4）	ルールは JA4 フィンガープリントと完全一致または前方一致で照合する。フィンガープリントは信頼できるイングレスが計算し、ヘッダーで転送する（`--ja4-header`）。Sibuna は計算しない	文書化されていない。ポリシーは、Anubis が計算して `X-Http-Fingerprint-JA4H` ヘッダーとして付加する HTTP フィンガープリント JA4H を参照できる	自身の TLS 終端で計算する。攻撃の詳細に表示される。カスタムルールで照合できる（等しい、等しくない）。組み込みのルールが、悪性 JA4 のインテリジェンスにあるフィンガープリントを拒否する	WAF のカスタムルールと分析で JA3 と JA4 のフィンガープリントを利用可能（*Bot Management* 付きの *Enterprise*）

表 5 機能比較、その 3：検査と状態

機能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
SQLi、XSS、RCE、トラバーサルの検査	Shield：タグ付きオートマトンと構造トークナイザー、ボディーの先頭 8 KB	-	OWASP のカテゴリーに対する意味解析エンジン。SSRF、	マネージドルールセット（*Pro* 以上）。OWASP Core Ruleset。攻撃

機能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
			XXE、CRLF も	スコア（*Business* 以上）
レート制限	クライアントごとの GCRA、ノードにローカル	-（システムの負荷を CEL ルールに公開）	IP、パス、セッションごと	プランにより *1 / 2 / 5 / 100* 個のルール。Enterprise 未満では IP のみのキー
禁止とレピュテーション	ハニーポットによる禁止。クラスターで共有するレピュテーションテーブルから読み込むレピュテーショントライ	DNSBL。Thoth による ASN と GeoIP（有償、クローズド）	IP グループとブラックリスト。脅威インテリジェンス（*Pro*）	IP リスト。マネージドリストとボットスコア（*Enterprise*）
フォレンジック	SQLite または PostgreSQL：インシデントの全文検索、ベクトルによるキャンペーンのクラスタリング	Prometheus のメトリクスのみ	コンソール付きの PostgreSQL の攻撃ログ	Security Events。Free ではサンプリング
複数ノード	すべてのノードがポリシーとレピュテーションのために共有する一つの PostgreSQL データベース。共有のシード。発行ノードに固定したチャレンジの検証	共有の署名鍵。共有の Valkey ストア	ホストごとに一つのスタック	グローバルなエニーキャスト

表 6 機能比較、その 4：運用

機能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
メトリクス	`/__sibuna/metrics` で Prometheus	別のポートで Prometheus	コンソールのダッシュボード	ダッシュボードと分析

機能	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
TLS の終端	-（イングレス）	-（イングレス）	**あり**	**あり**
管理コンソール	オプトイン。SID 0007 は Committed	-	**あり**	**あり**
ベンチマーク用ホストで測定	**あり**（第 VIII 部）	**あり**	-（Docker のみ）	-（ホスト型）

製品を選ぶ判断では、三つの違いが大部分の重みを持ちます。

- 作業がどこに行くか。Anubis と Sibuna Gate は、オリジンが何かをする前にクライアントに支払わせます。SafeLine と Cloudflare は先に検査し、選択的にチャレンジします。既定の姿勢は、許可してからふるいにかけることです。Sibuna Shield はその両方をこの順序で行います。検査は、すでに有効なセッションを持つリクエストも拒否できます。
- サーバーが何を記憶するか。Anubis はチャレンジの状態をストア（メモリ、bbolt、Valkey、S3）に保持し、Ed25519 で JSON Web Token に署名します。Sibuna は自己認証するレコードを発行し、解かれたタグだけを固定の表に記憶します。SafeLine は攻撃ログを PostgreSQL に保持します。Cloudflare はすべてを保持し、Free プランではサンプリングした範囲だけが見えます。
- 運用者が何を動かさなければならないか。Sibuna は一つのバイナリで、任意で SQLite ファイル、またはクラスターで共有する PostgreSQL データベースを使います。Anubis は一つの Go バイナリです。SafeLine は、少なくとも 1 コア、1 ギガバイトのメモリ、5 ギガバイトのディスクを持つ Linux ホスト上で、Docker デーモンの背後で動く七つのコンテナーです。Cloudflare は、DNS の変更とサブスクリプションです。

現在の製品の範囲　Sibuna は、受信側の TLS の終端をデプロイのプロキシに任せます。学習済みモデルによるボットのスコアリングや有償のシグネチャの公開も行いません。信頼できるイングレスが計算した JA4 フィンガープリントとは照合しますが、自らは計算できません（SID 0014、第 IX 部）。明示的に有効にするコンソールは、認証付きのダッシュボード、国情報の付加、ポリシーの編集、調査、クラスターの管理を提供します。国の検索とストレージの処理は、リクエストの分類とは別に実行します。SID 0007 は Committed で、コンソールはブラウザーでのワークフローに合格しています（SID 0010）。負荷の下で八つのダッシュボードを動かす正式な性能分離の判定はまだ測定していません。測定済みの内容は第 VIII 部に記録しています。

演習 2.1.　一つの検証器のソースをネイティブコードと WebAssembly にコンパイルすると、プロトコルの食い違いが減る理由を説明してください。それでも独立したテストや暗号のレビューが必要な性質はどれでしょうか。

演習 2.2. 比較表だけを使って、静的サイトに (a) 作業証明のゲート、(b) SQL インジェクションの検査、(c) クラスター全体での禁止を与える最小のデプロイ構成を書き出してください。それぞれの選択を決めたのはどの行でしょうか。
ヒント：「実行形態」と「複数ノード」の行から始めてください。

不変条件を説明してください。 Gate、Shield、任意の共有ストレージが、許可、検査、ポリシーの配布をどのように分担するかを説明してください。次に、顧客に引用する前に再検証したい比較表の行を一つ挙げ、どのように検証するかを述べてください。

第 III 部

暗号研究からの基礎

Sibuna の二つの作業証明の層を支えるコストモデルを導き、*Cohen-Pietrzak* の逐次作業証明をその安全性の議論とともに示し、チャレンジとセッションに鍵付きハッシュによる認証を使う根拠を説明します。前提と参考文献は *SID 0006* に記録しています。

Hashcash：第一層の数学

この章の学習目標 この章を読み終えると、ビット単位の Hashcash パズルの期待作業量、分散、裾の確率を述べ、サーバーの検証コストが攻撃者の優位を抑える理由を説明し、第二の層を必要とする二つの限界を説明できるようになります。

定義とコスト

チャレンジ文字列Cと難易度bビットに対する Sibuna の第一層のパズルは、SHA-256 $(C \,\|\, ":" \,\|\, \mathrm{dec}\,(N))$が先頭に$b$個のゼロビットを持つような$N \in \mathbb{N}$を見つけることです。nonce を十進数で表すのは意図的です。これにより、バイナリへのエンコード手順なしに、WebAssembly ソルバー、JavaScript による代替手段、ネイティブの検証器の間でワイヤーフォーマットが同一に保たれます。

難易度の単位は期待されるハッシュ試行であり、圧縮関数の呼び出しではありません。一つの入力が複数の SHA-256 ブロックにわたることがあります。接頭辞の事前計算によって、nonce あたりの作業量が変わることもあります。ワイヤー上の規則はあくまでダイジェストに関する述語です。実装が一致させるべきなのはステートメントと nonce のエンコーディングであり、推定した CPU コストではありません。

独立で理想的なダイジェストでは、試行回数Xは$p = 2^{-b}$の幾何分布に従います。その中央値は$\lceil \ln(1/2) / \ln(1-p) \rceil$で、$p$が小さいときはおよそ$2^b \ln 2$です。分散は$(1-p)/p^2$です。検証者は探索を行うのではなく、候補のダイジェストを一つ確認します。パース、チャレンジの認証、フィンガープリントの確認、使用済み状態への挿入、応答は追加の作業です。プリミティブの所要時間を逆算して、サーバーが耐えられる大量アクセスの量を求めることはできません。

探索の二つの限界

一つ目は分散です。探索が長いからといって、ワーカーが壊れているとは限りません。二つ目は並列性です。nonce の試行は互いに独立しているので、同時に実行できます。期待される総試行回数が同じでも、コアが多ければ探索の経過時間は短くなります。そのため、ハードウェアのスループットは許可のポリシーに影響します。公称のハッシュレートも単一のベンチマークも、あらゆる要求者の優位を抑えることはできません。

演習 3.0. 幾何分布の裾を使って、4.6×2^b回より多くの試行が必要になる確率を見積もってください。nonce 空間を複数のワーカーで分割しても期待試行回数が変わらないのはなぜでしょうか。
ヒント：経過秒数ではなく、総試行回数を数えてください。

逐次作業証明：第二層

この章の学習目標 Cohen-Pietrzak のラベル付け、それが逐次計算を強制する理由、検証者が確認する内容、条件付きのサンプリングの上限$(1-\alpha)^t$とその限界、耐量子の結果、証明者のメモリの上限、作業ビットの対応付けを支える較正データを理解します。

なぜ逐次作業か

逐次作業の構成は計算の間に依存関係を持ち込みます。後のラベルには、先に作られた値が必要です。これは総作業量とは別の量、つまり計算の深さを扱います。独立したパズルは依然として並列に実行でき、ハードウェアのハッシュのレイテンシーも依然として影響します。Sibuna は Cohen と Pietrzak が示したハッシュに基づく構成に従います。そのグラフは深さnで$2^{n+1}-1$個のラベルを持ちますが、ラベルの数が決まっていても、ブラウザーでのレイテンシーが決まるわけではありません。ラベルは複数の圧縮ブロックにわたることがあり、開示では部分木を再計算することがあり、スケジューリングもばらつきを加えます。オラクルモデルでの解析から得た安全性の主張を、具体的なすべての実装や敵対者についての無条件の主張に変えてはなりません。

構成

nを深さとし、長さn以下の二進文字列で名前を付けたノードを持つ完全二分木を考えます（根はε、子は$v0$と$v1$、葉は長さn）。各ノードvは 32 バイトのラベル$\ell_v = H(\chi, v, \text{parents}\ (v))$を持ちます。ここで$\chi$はチャレンジのステートメント、$H$は$\chi$を一度だけ吸収した SHA-256 です（Hashcash ソルバーと同じ接頭辞の工夫です）。

- 内部ノードは二つの子をハッシュします：$\ell_v = H(\chi, v, \ell_{v0}, \ell_{v1})$。
- 葉$u = u_1 \ldots u_n$は、根からの経路が右に曲がったところすべてについて、祖先の左の兄弟をハッシュします：$\ell_u = H(\chi, u, \{\ell_{u_1 \ldots u_{i-1} 0} : u_i = 1\})$。

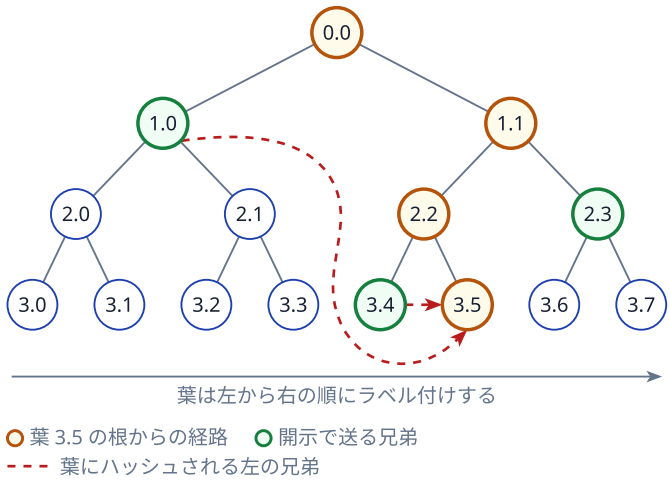


図 3 深さ 3 の木。葉 3.5 の開示では、葉のラベルと三つの兄弟のラベルを送ります。破線の辺は、ラベル付けを逐次的にする左の兄弟への依存です。

計算を逐次的にするのは左の兄弟の辺です。葉uは、その左にあるすべての部分木が完成するまでラベルを付けられません。そのため深さ優先の帰りがけ順の走査が強制され、根のラベル$\varphi = \ell_\varepsilon$は$2^{n+1} - 1$個すべてのラベルに順に依存します。

開示。φを計算した後、証明者は Fiat-Shamir によってt個の葉$\gamma_i = H(\chi \,\|\, \varphi \,\|\, i) \bmod 2^n$を導き、それぞれについて葉のラベルと、根への経路上のn個の兄弟のラベルを送ります。検証者は、親である兄弟から葉を再計算し、経路に沿って上へハッシュし、φに到達することを確認します。検証者のコストは、有界な作業領域での$O(tn)$回のラベル計算です。一回のラベル計算は複数のハッシュブロックにわたることがあります。ラベルを数えることは、圧縮関数の呼び出しを数えることではありません。

> **実装：`posw::verify_opening`**　開示された葉一つを兄弟のラベルから再計算し、根までの経路をたどります。`crates/pow/src/posw.rs`（クレート `sibuna-pow`）にあります。

```
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`は深さnから深さ1へ並ぶ。逆順にすると深さ1..=nと対応する。
    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
}
```

サンプリング、健全性、その限界

独立に検出できる不正な割合αを持つ固定したコミットメントを考えます。各開示が一様かつ独立にサンプリングするなら、t回の開示で不正な位置をすべて見逃す確率は$(1-\alpha)^t$です。この初等的な計算は、開示が多いほど役立つ理由を説明します。しかし、これだけでは逐次作業プロトコルの安全性の証明にはなりません。

例題：条件付きのサンプリングの上限 不正な割合を$\alpha = 0.1$に固定し、独立な開示を$t = 16$回行うと、不正な位置をすべて見逃す確率は$0.9^{16} \approx 0.185$です。開示が 32 回なら約 0.034 です。この前提では、サンプルを選ぶ前にコミットメントが固定されています。証明者が多数の根を試せる場合、どのコミットメントを明かすかを選べる場合、失敗を相関させられる場合、この計算はその戦略を考慮していません。

参照した逐次作業の構成には、オラクルモデルでの安全性の解析があります。Sibuna のドメイン分離、パラメーターの対応付け、切り詰め、実装は、それでもなお構成の前提に照らして確認しなければなりません。Fiat-Shamir によるサンプリングは、コミットメントから開示を再現可能にします。再現可能であるだけでは、総当たりによる試行への耐性は証明されません。不正が決して安くならないという無条件の主張はしません。

演習 3.1. 攻撃者が独立なコミットメントをg個試し、それぞれが確率qで受理されるとします。少なくとも一つが受理される確率を導いてください。その式にはどのコストが欠けているでしょうか。
ヒント：まずg個すべてが失敗する確率を計算してください。

証明者のメモリ

素朴な証明者はすべてのラベル（$2^{n+1}-1$個）を保存します。Sibuna の証明者は、最初の走査の間は現在の経路の左の兄弟のラベルのスタック（$n+1$個のラベル）を持ち、上位$m = \min(n, 10)$段（$2^{m+1}-1$個のラベル、最大 64 KB）だけを保持します。葉を開示するときは、その葉の深さmの祖先の部分木を$2^{n-m+1}-1$回のハッシュで再計算し、保持した段からスタックを初期化します。証明者の総コストは$2^{n+1}-1+t\,(2^{n-m+1}-1)$回のハッシュで、出力する

証明を含む作業領域全体は 100 KB 未満です。ブラウザーのタブとネイティブのテストは同じ `Workspace` 型（`sibuna_pow::posw::Workspace`）を共有します。

パラメーターは所要時間ではない

Sibuna は$n = b - 3$を使って作業ビットを深さに対応付け、サポートする区間$[4, 24]$に収めます。これは作業の尺度に関する取り決めであり、アルゴリズム間で実時間が等しいことを保証するものではありません。葉は複数の親のラベルをハッシュすることがあり、Hashcash は nonce を繰り返し試します。コンパイラー、ブラウザー、プロセッサー、パラメーターの選択はすべて、両者の相対的なコストに影響します。コミット済みのプリミティブのベンチマークは検証を測るもので、ブラウザーでの解答のレイテンシーを示すものではありません。
証明のサイズは$32(1 + t(n + 1))$バイトです。深さ 13 で開示が 16 回なら、$32(1 + 16 \times 14) = 7200$バイトです。開示の回数を増やすと、木を構築する主要なコストが同じでも、証明と検証者の作業の両方が増えます。

演習 3.2. 深さ 17、開示 16 回での証明のサイズを計算してください。次に開示の回数を 2 倍にしてください。どの項が 2 倍になり、どの項が変わらないでしょうか。

構成の選び方

比べるべきはアルゴリズムの評判ではなく、資源の予算です。証明者のメモリ、検証者の最悪時の作業量、証明のバイト数、主張する性質に必要な前提を記録してください。メモリハードな探索は役立つことがありますが、高価な作業を繰り返さなければならない検証器は、攻撃者にサーバーの資源を消費させる別の場所を与えます。公開検証可能な遅延の構成は、秘密を共有するゲートとは別の信頼モデルに向いています。
Sibuna は、安価なハッシュによる検証と、ハッシュに基づく逐次作業の選択肢を保持しています。後者は証明のバイト数が多く、パラメーターをより慎重に解析する必要があります。どちらを選んでも、高価な検証に入る前に不正な形式の証明を拒否する必要はなくなりません。

不変条件を説明してください. ラベルがコミットメントに属することを証明する認証木と、ラベルの計算方法を制約する依存の辺との違いを説明してください。

対称鍵による認証：チャレンジとセッション

この章の学習目標 発行者と検証者が同じ信頼ドメインにあるとき鍵付き擬似ランダム関数が正しいプリミティブである理由を説明し、128 ビットのタグに対する偽造の上限を述べ、状態を持たないチャレンジレコードと使用済み集合の上限を説明し、鍵スケジュールを読み解きます。

発行者と検証者は同じ

デジタル署名では、公開鍵を持つ誰でも検証できます。検証者に発行させてはならない場合には、これが正しい道具です。Sibuna の検証者は発行者そのものです（または一つのシードを共有するクラスターです）。擬似ランダム関数から作るメッセージ認証コードは、その信頼ドメインの中での認証を提供します。公開検証は提供しません。MAC 鍵を持つ者は誰でもトークンを発行できます。ベンチマークの章では、実装したトークンの操作を比較します。

補題 3（偽造の上限） 鍵付きモードの BLAKE3 を PRF としてモデル化し、タグを 16 バイトとすると、q回の検証クエリを行う敵対者が有効なトークンを偽造する確率は高々 $q \cdot 2^{-128} + \varepsilon_{\mathrm{PRF}}$です。毎秒 100 万クエリでも、毎秒$2^{-108}$です。

長期的にはもう一つの帰結が重要です。Ed25519 は楕円曲線群の離散対数に依拠しており、これは Shor のアルゴリズムで破られます。鍵付きハッシュはハッシュ関数だけに依拠しており、量子アルゴリズムによる高速化は高々二次です。Sibuna は、発行能力を持たない検証者を必要とするデプロイのために、Ed25519 を選択肢（`--token-scheme ed25519`）として残しています。

鍵スケジュール

32 バイトのマスターシード一つ（`--secret-file`、`SIBUNA_SECRET`、または新たに生成した乱数値から得ます）から、鍵付き BLAKE3 とドメイン文字列によって用途ごとの鍵をすべて導きます。トークン鍵、チャレンジ鍵、Ed25519 鍵、フィンガープリント鍵、そして要件チケットを封印する要件鍵です（第 IV 部）。

実装：`Keys::derive` シードから用途ごとの五つの鍵を導きます。`crates/challenge/src/keys.rs`（クレート `sibuna-challenge`）にあります。`Keys` と `Seed` は破棄されるときにバイトをゼロにし、`Debug` 出力は伏せ字になります。

```
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())
}
```

クラスターでは、`Keys::bind_node` がチャレンジ鍵を、その鍵の下で `sibuna/challenge-node/v1` とノード ID に対して計算した鍵付き BLAKE3 で置き換えます。これにより、チャレンジは発行し

たノードに戻らなければなりません。一方、セッショントークンと要件チケットはすべてのメンバーで有効なままです。
ドメイン分離は、用途をまたぐ偶発的な再利用を防ぎます。マスターシードが漏れれば、導出したすべての鍵が漏れます。クラスターが合意すべきなのはシード一つだけです。

状態を持たないチャレンジ

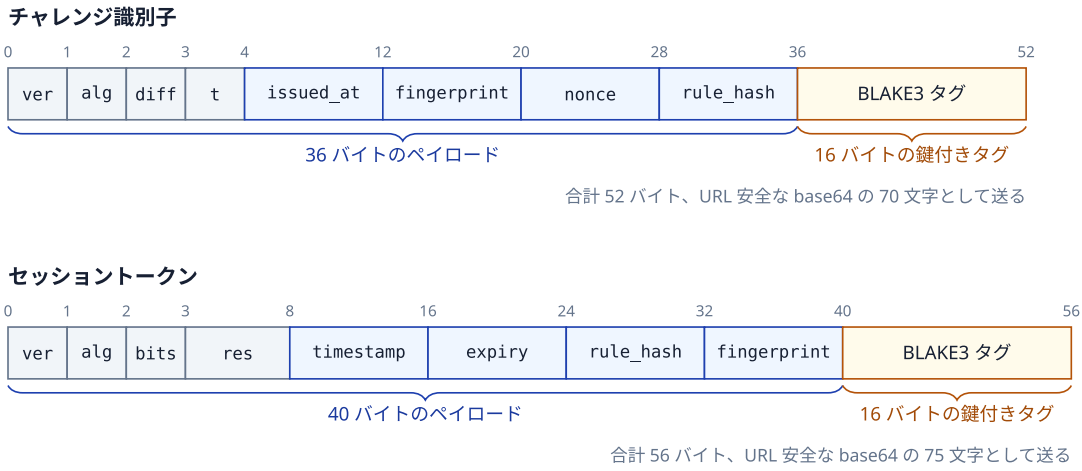


図 4 チャレンジ識別子とセッショントークンのワイヤーフォーマット。各レコードの上の数字はバイトオフセットです。

チャレンジ識別子は自己認証するレコードです。バージョン、アルゴリズム、難易度、開示数、発行時刻、クライアントの鍵付きフィンガープリント、PRF で導いた nonce、チャレンジを要求したポリシールールのハッシュを持ち、その後にチャレンジ鍵による 16 バイトのタグが続きます。発行しても何も書き込みません。検証ではレコードをデコードし、タグを定数時間で確認し、経過時間をチャレンジの TTL と照合し、フィンガープリントを送信元のクライアントと照合し、その後で初めて証明を調べます。

補題 4（状態は支払われた作業に比例してのみ増える） Sをデーモンが記憶するチャレンジタグの集合とします。タグがSに入るのは、そのタグに対する有効な証明が受理された後だけで、チャレンジの TTL が過ぎると出ていきます。したがって、その期間に受理される異なる解答の速度の上限をrとすると、$|S| \le r \cdot \mathrm{TTL}$です。無償のチャレンジ取得ではタグは割り当てられません。実装は容量も固定しており、満杯のときは有効な解答を拒否することがあります。これは状態の上限であり、あらゆる敵対者の計算戦略に対する下限ではありません。

使用済み集合は、16 バイトのタグに対する Robin Hood オープンアドレス法の 16 個のシャードです（第 V 部）。フィンガープリントはクライアントのアドレスと User-Agent の鍵付きハッシュです。連結によって二つの入力が衝突しないように長さで区切り、他のクライアントのフィンガープリントをオフラインで計算できないように鍵を付けています。

演習 3.3. トークンのペイロードは、作業レベル、`timestamp`、`expiry`、`rule_hash`、`fingerprint` を持ちます。各フィールドを取り除いたら何が防げなくなるかを説明し、タグが五つすべてを覆わなければならない理由を説明してください。

不変条件を説明してください. 同僚が、すべてのトークンを Ed25519 に切り替えようと提案します。「そのほうが強いから」というのです。信頼ドメインの議論、コストの違い、耐量子性の観点を説明してください。

第 I V 部

Sibuna プロトコル

実装どおりのワイヤープロトコルを規定します。中間ページの往復、状態を持たないチャレンジレコード、二つの解答形式、検証の順序、一回限りの使用の強制、セッション *Cookie* です。

チャレンジの往復

この章の学習目標　この章を読み終えると、ブラウザーの最初のリクエストからセッションの発行までをたどり、関係するすべてのエンドポイントと JSON フィールドを挙げ、チャレンジが保護されたパスのポリシー判断を運ぶ理由を説明できるようになります。

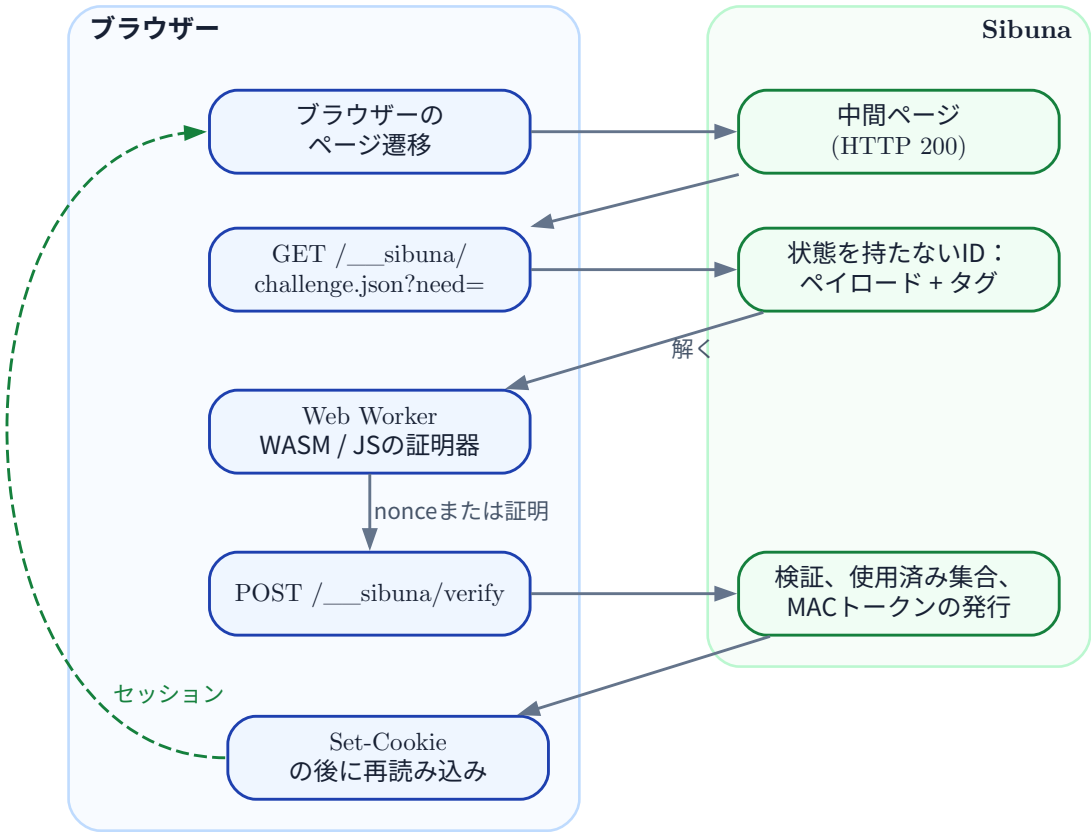


図 5 チャレンジの往復。左側の手順はブラウザーで、右側の手順は Sibuna で実行されます。

1. 有効なセッションを持たないページ遷移はポリシーエンジンに達し、CHALLENGE に分類されます。リクエストが text/html を受け付けるので、デーモンは 200 と中間ページ、そしてヘッダー X-Sibuna-Status: CHALLENGE で応答します。HTML を受け付けない API クライアントは 401 と、すぐに取得できる challengeURL を含む JSON ボディーを受け取ります。どちらの場合も、応答は要件チケットを運びます。これは、この判断が要求するチャレンジ（アルゴリズム、作業ビット、開示数）とルールのハッシュを、専用に導出した鍵による鍵付き BLAKE3 タグでこのクライアント向けに封印したもので、チャレンジの有効期間一つ分だけ有効です。
2. 中間ページは GET /__sibuna/challenge.json?path=<original URL>&need=<ticket>を取得します。有効なチケットがあれば、それがチャレンジを決めます。作業量はそれを要求した判断によって固定され、報告された URL と取得リクエスト自身のヘッダーから再計算されることはありません。取得リクエストのヘッダー（Accept、Sec-Fetch-*）はページ遷移のものとは異なるからです。有効なチケットがない場合（URL を無視する API クライアントや、有効期間を過ぎて開いたままのページ）、サーバーは報告された URL を評価します。URL はリクエスト行を読むのと同じ関数でパスとクエリに分けられ、パスは許可の判断で使う正規形に変換されます（SID 0009）。8 KiB より長い URL は切り詰めずに 414 で拒否し、正規形を持たないパスは 400 で拒否します。チケットは助言にすぎません。許可の判断では依然としてセッションの作業レベルを確認するので、再送や偽造したチケットは保有者に作業を課すだけです。
3. 応答はチャレンジレコードです。

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

   hashcash では、difficulty はビット数で、challenges は 0 です。
4. Web Worker が WebAssembly モジュールで、あるいは WebAssembly が使えなければ JavaScript の証明器で（第 VII 部）これを解き、次のいずれか

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

   または

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

   を POST /__sibuna/verify に送信します。
5. 成功すると、サーバーは Set-Cookie ヘッダー付きの 200 {"status":"ok"}で応答し、ページは元の場所を再読み込みします。以後のリクエストはすべて Cookie を運び、ポリシーの評価の前に許可されます。

パスがチャレンジとともに送られる理由 中間ページは保護された URL で配信されるので、ブラウザーは自分がどこにいるかを知っています。パスを送ることで、サーバーはルールごとの難易度とアルゴリズムを選べます（決済のルートは 20 作業ビットの逐次作業を要求し、ブログは 16 ビットの Hashcash を要求する、といった具合です）。さらにルールのハッシュをトークンに結び付け、X-Sibuna-Rule-Hash がそれを上流に伝えます。

状態を持たないチャレンジレコード

この章の学習目標　チャレンジのエンコーダーを読み、各フィールドの目的を理解し、ホットパスでエントロピーのシステムコールを使わずにチャレンジごとの nonce を導く方法を確かめます。

実装：`Coordinator::create_challenge_with_spec`　36 バイトのペイロードを作り、チャレンジ鍵でタグを付け、52 バイトを base64url で 70 文字の識別子にエンコードします。`crates/challenge/src/coordinator.rs`（クレート `sibuna-challenge`）にあります。

```
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)),
    }
}
```

識別子は Encoded<70>です。テキストはちょうどその長さの配列に置かれるので、チャレンジの発行でメモリを確保することはありません。ChallengeSpec::parameter は作業ビットを層ごとの単位に対応付けます。Hashcash のビット（最大 64）か、PoSW の深さ（difficulty - 3 を 4 から 24 の範囲に収めた値）です。
注目すべき点が三つあります。

- 適応的な難易度。コーディネーターは、チャレンジの発行率の指数加重移動平均を三つのアトミック変数で保持します。毎秒 50 チャレンジという基準を超えると、発行率が 2 倍になるごとに作業ビットを一つ加え、最大で六つまで加えます。大量アクセスは指数的に多くを支払い、静かなトラフィックは基本コストのままです。
- *nonce* は *PRF* の出力です。チャレンジ鍵を鍵とし、カウンター、時刻、フィンガープリントから計算します。nonce は一意（カウンターによる）で予測不能（鍵による）なので、敵対者は発行されていない識別子の解答を事前に計算できません。また、ホットパスでオペレーティングシステムにエントロピーを求めることもありません。
- 難易度はタグの中に含まれます。クライアントがレコードを書き換えて難易度を下げることはできません。証明を調べる前に、タグの確認で失敗するからです。

検証の順序と一回限りの使用

この章の学習目標 安価な確認が証明より先に行われる理由、証明が有効になってから初めてチャレンジを使用済みにする理由、同じ解答の二つの同時提出がどう解決されるかを理解します。

実装：Coordinator::verify_and_mint 識別子をデコードして認証し、経過時間とクライアントへの結び付きを確認し、記録された層の証明を検証し、タグを使用済みとして記録し、セッショントークンを発行します。crates/challenge/src/coordinator.rs（クレート sibuna-challenge）にあります。

```
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,
    })
}
```

この順序は意図したものです。タグ、経過時間、結び付きは定数時間の確認で、ハッシュ計算の前に不正なデータを拒否します。誤った証明でチャレンジを消費してはならないので、使用済み集合への書き込みは check_solution が成功した後だけです。同じ有効な解答を二つのスレッドが検証すると、mark_spent で競合します。シャードのロックがそれらを直列化し、ちょうど一方が勝ち、もう一方は VerifyError::DoubleSpendAttempt を受け取ります。条件 issued_at > now + 60（CLOCK_SKEW_SECONDS は 60）は、時計が 1 分より進んでいるノードが発行したレコードを拒否します。そうしなければ、チャレンジの寿命が延びてしまいます。
エラーは Elm 形式の診断（第 X 部）とともに 400 としてクライアントに返ります。各 VerifyError は sibuna_core::errors::ErrorKind に変換され、その説明が応答のボディーになります。エンドツーエンドテストは、応答のボディーに DOUBLE SPEND、FINGERPRINT MISMATCH、WRONG SOLUTION TYPE、MALFORMED CHALLENGE が含まれることを確認します。

セッション Cookie

この章の学習目標　鍵付きハッシュのトークンを読み、Cookie の属性を理解し、クライアントへの結び付きを説明します。

実装：token::mac::mint と token::mac::verify　40 バイトのペイロードをシリアライズし、それに対する鍵付き BLAKE3 の先頭 16 バイトを付け加え、定数時間の比較で検証します。crates/challenge/src/token/mac.rs（クレート sibuna-challenge）にあります。Coordinator::verify_cookie がクライアントのフィンガープリントを渡して呼び出します。

```
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);
    // 定数時間：途中で抜けると、タグの何バイトが正しかったかを偽造者に教えてしまう。
    if !ct::eq(&tag(key, &bytes), given) {
        return Err(TokenError::InvalidTokenSignature);
    }
    let payload = Payload::from_bytes(&bytes)?;
    payload.check(now, expected_fingerprint)?;
    Ok(payload)
}
```

タグはペイロードをパースする前に確認します。別のバージョンのペイロードは、本物であると分かった後で初めて `TokenError::InvalidEncoding` で拒否されます。偽造したペイロードがパーサーに届くことはありません。
Cookie は次の形で出力されます。

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

`HttpOnly` は Cookie をページのスクリプトから隠します。`SameSite=Lax` はトップレベルのページ遷移を許しつつ、クロスサイトでの再利用を防ぎます。`Secure` は、デーモンの前段で TLS を終端する場合に `--secure-cookie` で付きます。ペイロードはアドレスと User-Agent の鍵付きフィンガープリントを運ぶので、別のマシンにコピーした Cookie は `TokenError::TokenBoundAddressMismatch` で失敗します。エンドツーエンドテスト「the cookie is bound to the client identity」がまさにその経路を確かめます。
ペイロードは、バージョンのバイトと、保有者が支払った作業レベルで始まります。作業レベルは方式（Hashcash または PoSW）と実際に解いた作業ビットで、負荷に応じた上乗せも含みます。セッションがチャレンジを免除されるのは、そのレベルがルートの要求に達するときだけです。要求は、チャレンジ自身と同じ範囲の制限を通してルールの難易度から計算します。レベルには名前ではなく順序があります。24 ビットで得た Cookie は 16 ビットのルートもすべて満たしますが、24 ビットのルートに 16 ビットの Cookie を示すと、より強い中間ページに回され、新しい Cookie が古いものを置き換えます。`rule_hash` は引き続き、チャレンジを発行したルールを監査のために識別するものです。上流には伝えますが、許可の判断には使いません。WAF の検出と明示的な拒否は、セッションによって免除されることはありません。エンドツーエンドテスト「a session earned on a cheaper route does not admit a route that demands more work」が両方向を確かめます。

> **転送されたアドレス**　イングレスの背後では、クライアントのアドレスは `X-Forwarded-For` で届きます。Sibuna がこのヘッダーを信頼するのは `--trust-forwarded` を指定したときだけです（forward-auth モードでは暗黙に有効です）。インターネットに直接公開するリバー

スプロキシでは無効にしておかなければなりません。そうしないと、どのクライアントも自分の結び付きとレート制限のバケットを自由に選べてしまいます。

演習 4.1. 一つの NAT アドレスの背後にある、User-Agent が同一の二つのブラウザーのシーケンスを描いてください。両者が共有する確認と共有しない確認はどれでしょうか。その NAT の背後にいる攻撃者が、もう一方のブラウザーの Cookie を再利用するには何が必要でしょうか。

不変条件を説明してください. デーモンが未解決のチャレンジを「記憶」しない理由と、それが手抜きではなくセキュリティ上の特長である理由を、運用者に説明してください。

第 V 部

Rust によるゼロアロケーションのエンジニアリング

一つの固定バッファーを通して、接続を受け付けてから応答するまでを追います。また、多くの接続スレッドがリクエストの経路でヒープ割り当てをせずに状態を共有するための並行データ構造を調べます。メモリ安全性のために本来なら専用の仕組み（読み手のカウント、アリーナ、ディスクリプターの表、後始末の連鎖）が必要になる箇所では、*Rust* の所有権が代わりにそれを提供する様子を示します。

接続のループ

この章の学習目標　この章を読み終えると、64 KiB のバッファー一つが keep-alive 接続全体にどう使われるか、ヘッダー部とボディーをコピーせずにどう見つけるか、なぜ各接続が上限のある専用スレッドを持つのか、プロキシする側のクライアント接続を開いたままにするために、プロキシがオリジンのフレーミングをどう知るのか、そしてこれらの保証のうちどれを借用チェッカーが証明してくれるのかを説明できるようになります。

接続ごとに一つのスレッド、ただし上限付き

受付ワーカー（`--workers`、既定では CPU ごとに一つ、最大 64）は待ち受けソケットを共有します。受け付けた接続はそれぞれ、1 メガバイトのスタックを持つ専用のスレッドに渡され、受付ループはすぐに `accept` に戻ります。接続を終えたスレッドは待機して次の接続を引き受けます。接続ごとにスレッドを生成して終了させると、他のスレッドが停滞したからです（第 VIII 部、SID 0002）。別の案である、受付スレッドで接続を最後まで処理する方式が最初の設計でした。クライアントが一つなら良い測定値でしたが、64 では悪い値でした。そのうち 60 がワーカーの 256 リクエストの上限が終わるのを待ち、待つクライアントが増えるごとに裾のレイテンシーが伸びたからです。接続ごとにスレッドを持たせると、64 接続での p99 は、製品全体のどのワークロードでも 1 ミリ秒未満、305 us から 857 us に収まります（`benchmarks/results/product-latest.md`、第 VIII 部）。

接続スレッドの数は`--max-connections`（既定で 1,024、最大 8,192）で制限されます。上限を超えると、受付ループは新しいソケットに `503 Service Unavailable` で応答し、何も生成せずに閉じ、その事象を `sibuna_overloaded_total` に数えます。

実装：`accept_loop`　接続を受け付け、それぞれを接続数の上限と照合し、タスクプールから専用のスレッドで起動します。`cmd/sibuna/src/server.rs`（クレート `sibuna`）にあります。

```
fn accept_loop(server: &Arc<Server>, listener: &TcpListener) {
    while !server.is_stopping() {
        let stream = match listener.accept() {
            Ok((stream, _)) => stream,
            // ... ディスクリプターかメモリが尽きた：スピンせずに1 ms待つ
        };
        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);
        });
        // ... 起動に失敗したら枠を返し、過負荷として数える
    }
}
```

`Counted` はガードで、その `Drop` は、スレッドが戻り値で終わってもパニックで終わっても、接続が上限の中で占めていた枠を返します。`503` の経路では、ソケットの自分の側を閉じる前に、クライアントがすでに送ったデータを読み捨てます。未読のバイトが残ったままソケットを閉じるとリセットで応答することになり、そのリセットは `503` を破棄してしまうからです。スレッドは `TaskPool`（`cmd/sibuna/src/tasks.rs`）から取り出します。これは join 可能なスロットの固定表です。終了したスレッドは join されるまでスタックを保持するので、スロットは再び取り出されるときに join され、新しいタスクはカーソルを進めるのではなく、終了済みのスロットのうち最も小さいものを取ります。カーソルでは、表を一周した後でないと終了済みのスロットに戻らず、何千もの終了済みスレッドがメモリに残ります。そしてメモリは開いている接続の数ではなく、処理した接続の数に応じて増えてしまいます。二つ目のガード `Finished` はタスクの終了時にスロットを空きとして記録するので、パニックした接続もスロットを解放します。

メモリも同じように上限を持ちます。接続スレッドは 64 KiB の読み込みバッファーと 16 KiB の書き込みバッファーをスタックに、40 KiB のプロキシ作業領域をヒープに持ち、作業領域は接続の開始時に一度だけ確保します。したがって最悪の場合は、クライアントが開けるソケットの数の関数ではなく、既知の数値になります。アイドル接続回収スレッド（後述）は送信をやめた接続を閉じるので、遅いクライアントが `--idle-timeout` を超えてスレッドを占有することはできません。

アロケーターの選択

Rust 1.100 では `Allocator` API の中核と、`Vec` および `Box` のアロケーターパラメーターが安定化されました（リリースノート）。私たちはプロセスのアロケーターを置き換えずに、個々のコレクションのアロケーターを選べます。この API によってリクエスト経路の割り当てがさらに減るわけではありません。ウォームアップ後のリクエストはすでにメモリーを割り当て

ず、cmd/sibuna/tests/steady_state.rs と crates/net/tests/steady_state.rs で確認しています。プロキシは接続ごとに三つのバッファーを一度だけ割り当て、再利用します。レートリミッターと使用済み証明集合もテーブルを一度だけ割り当てます。ゼロ初期化された割り当てでは、未使用のページを常駐させずに済み、crates/store/tests/resident.rs で確認しています。

アロケーターの変更を調べる際には、まずポリシースナップショットの構築とコンソールの JSON 解析を測定します。一時的なコレクションはリクエスト経路の外でメモリーを割り当てますが、コレクションのアロケーターが要素の割り当てまで自動的に管理するわけではありません。JSON ツリーはエスケープを含む文字列も Cow<str> を通して所有し、その所有形式は String です。Rust 1.100 の String にはアロケーターパラメーターがありません。Arc::new_in もこのリリースでは不安定なままです（API ドキュメント）。公開したエンジンは最後の読み手が解放するまで Arc で共有されるため、再構築時に古いエンジンが使うメモリーをリセットできません。アリーナでも、この生存期間とパーサーの入力上限を維持する必要があります。

私たちはコンソールの WebAssembly モジュールにある既存のグローバルアロケーター WipeOnFree を cmd/console-ui/src/abi.rs に維持します。これは解放するブロックを消去し、再割り当て後の古いブロックも消去します。コレクションに System を直接選ぶと、この消去を迂回します。作業証明の WebAssembly モジュールにはヒープアロケーターがなく、Argon2 も呼び出し側が所有する作業領域をすでに再利用しています。そのため、測定した負荷が変更を正当化するまでは、既存のアロケーターと Rust 1.94 との互換性を維持します。候補は、消去、上限付きの入力処理、割り当てのないリクエスト経路を保ったうえで、レイテンシーか常駐メモリーを改善する必要があります。

接続ごとに一つのバッファー

リクエストの経路での動的なメモリ割り当ては、どれもサービス拒否のてこになります。何日もかけて進む断片化、スレッド間でのアロケーターのロック競合、割り当てを最大化するリクエストを作る攻撃者による増幅です。Sibuna の規則は絶対的です。分類、検証、プロキシへの受け渡しは、ヒープ割り当てなしで実行します。接続ハンドラーは、ソケットが生きている間、64 KiB の読み込みバッファーと 16 KiB の書き込みバッファーを所有します。それ以外はすべて、それらを指すスライスです。

実装：connection::serve　接続をアイドル接続回収スレッドに登録し、バッファー、プロキシ作業領域、エンジンのキャッシュを一度だけ用意し、最大 4,096 件のリクエストを処理します。cmd/sibuna/src/connection.rs（クレート sibuna）にあります。

```
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_head は空行が現れるまでバッファーを満たします。探索するのは前回の探索以降に届いたバイトだけで、16 KiB を超えるヘッダー部は 431 で拒否します。parse_request（クレート sibuna-http）は Request<'a>を生成します。そのメソッド、パス、クエリ、ヘッダー、Cookie はバッファーのスライスです。宣言されたボディーは収まる分まで読み込み、ヘッダー部の後ろをスライスします。その後 reader.consume(head_len + body_len)がリーダーを進めるので、次の keep-alive リクエストはきれいに始まります。リーダーは、場所が必要なときだけ未読のバイトを先頭に移します。チャンク形式のボディーは、ヘッダー部の後ろでその場でデコードします（SID 0009）。各ステップのフレーミングをバッファーから切り取るので、デコードした先頭部分は検査のために連続したままです。内部ルートは長さで区切られ、接続を開いたままにします。プロキシしたリクエストも、オリジンの応答がフレーミングされていて、ボディーが完全にバッファーされていれば、接続を開いたままにします。プロキシがどう判断するかは次の節で示します。

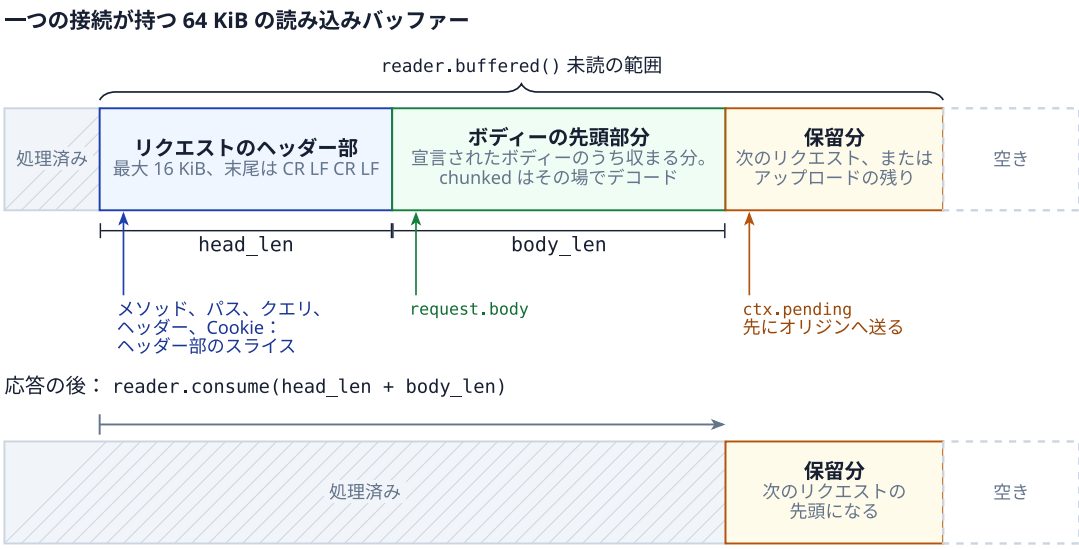


図 6 一つのリクエストを処理している間の接続の読み込みバッファー。リクエストのどのフィールドも、このバッファーのスライスです。

```rust
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)
}
```

借用チェッカーが証明すること　リクエストのスライスが有効なのは、次にバッファーを満たすときまでです。Sibuna はこの規則をコメントではなく型として表します。リクエストは `Request<'a>` で、`'a` はリーダーのバッファーの共有借用です。バッファーを満たしたり消費したりするには `&mut reader` が必要で、そこから借用した `Request` がまだ使われている間、コンパイラーはこれを拒否します。`serve_one` では、リクエストは `dispatch` にムーブされ、その呼び出しが戻った後で初めて `reader.consume` を実行できます。逆の順序で書くとコンパイルできません。

同じ規則で、一見奇妙な二つの選択も説明できます。一つ目は、`serve_with_body` がヘッダー部を二回パースすることです。チャンク形式のボディーをバッファーするとバッファーの中でバイトが移動するので、最初のパースはボディーをバッファーする前に借用を終えなければなりません。二回目のパースは落ち着いたバッファーを借用します。パースは決定的で、Cookie 付きのブラウザーのリクエストで約 133 ns かかります（`benchmarks/results/primitives-latest.md` の `zero_copy_request_and_cookie`、132.5 ns）。そのため二回目のパースのコストは小さく、同じ判定が得られます。二つ目は、プロキシがパース済みのリクエストと保留中のバイトを借用しているため、接続バッファーを満たし直してはならず、クライアントからの続きのバイトを専用のアップロードバッファーに読み込むことです。読んだものを返すことはできないので、クライアント接続を維持するのはボディーが完全にバッファーされたときだけです。

プロキシでのヘッダー部の書き換え

ヘッダー部はそのまま転送するのではありません。write_request_head（crates/net/src/request.rs、クレート sibuna-net）は、リクエスト行と各ヘッダーをオリジン向けのライターに出力し直します。その際、ホップごとのフィールド（Connection、Keep-Alive、Proxy-Connection、Transfer-Encoding、TE、Trailer、Upgrade、プロキシ認証のフィールド）、受信した転送用または監査用のフィールド（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、そしてリクエストの Connection ヘッダーが指名するすべてのフィールドを取り除きます。その後、監査用のフィールドの組を付け加えます。ボディーのフレーミングは生成するもので、コピーはしません。

```
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-Length は出力し直します。接続バッファーの中で終わるチャンク形式のボディーは、その場でデコードし、長さを付けて送ります。それより長いものは Transfer-Encoding: chunked と告げ、一回の読み込みごとにチャンク化し直します（SID 0009）。各読み込みはプロキシの 16 KiB のアップロードバッファーに入り、そこでその場でデコードされ、正規の一つのチャンクとして送り出されます。そのため、オリジンがクライアントのチャンクサイズ、拡張、トレーラーを見ることはありません。監査用のフィールドの組は次のとおりです。

```
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 | ...
```

バッファーより大きいボディーは、応答をストリームで返す前に、クライアントからオリジン向けのライターへ 16 KiB ずつ中継します（relay_body）。crates/net/src/request.rs の単体テスト proxy_head_rewrite_drops_hop_by_hop_headers_and_injects_audit_fields は、書き換えが偽装された X-Forwarded-For を取り除き、監査用のフィールドを挿入することを確認します。
プロキシの三つのバッファー、つまりオリジンの読み込みバッファー（応答のヘッダー部一つ、16 KiB）、オリジンの書き込みバッファー（8 KiB）、アップロードバッファー（16 KiB）は、

接続が一度だけ確保してすべてのリクエストで再利用する Workspace にあります。作業領域によって、それらの寿命は一回のやり取りではなく接続の寿命になり、オリジンのリーダーとライターは一回のやり取りの間だけそれらを借用します。

オリジンの応答の中継

最初のプロキシは、オリジンが閉じるまでオリジンのバイトをストリームし、その後クライアントを閉じていました。パースなしで正しくフレーミングできますが、プロキシするリクエストごとに新しい TCP 接続が必要になります。負荷生成器の下では、その代償が TIME_WAIT にある何万ものソケットとして現れ、ループバックではエフェメラルポートを使い果たしました。現在のプロキシはオリジンのヘッダー部（最大 16 KiB）を読み、RFC 9112 の規則でボディーを分類します。

実装：parse_response_head　ステータス、四つのフレーミングのいずれか（HEAD、1xx、204、304 ではボディーなし。Content-Length。chunked 転送コーディング。接続の切断で区切るもの）、オリジンが接続を維持するかどうかを返します。crates/net/src/response.rs（クレート sibuna-net）にあります。

```
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 })
}
```

パーサーは、プロキシが中継してはならないものを拒否します。不正なステータス行、トークンでないフィールド名、値の中の制御バイト、矛盾する長さ、繰り返された、または最後でない chunked コーディング、Content-Length や Transfer-Encoding を指名する Connection フィールドです。その作業量はヘッダー部の長さに比例し、決してパニックしません。
ヘッダー部は、オリジンの Connection ヘッダーを Sibuna 自身の判断で置き換えたうえでクライアントに出力し直し、ボディーは正確に中継します。長さがあれば copy_exact、chunked コーディングならチャンクごと（サイズ行、データ、トレーラー）、旧来の場合は copy_remaining です。クライアントを閉じるのは旧来の場合だけです。どの中継も一つのコピー手順の上に作られています。送り元がすでに持っているものは何でも転送し、何も持っていないときだけ宛先をフラッシュしてから一回読み込みます。そのため小さな書き込みの流れは生成されたとおりに届き、大量転送では読み書きのたびにバッファー一杯分が動きます。

```
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 })
```

8 フィールドのオリジンのヘッダー部のパースには 322 ns、1 KiB の応答を返す keep-alive の GET をループバックでプロキシするには 29.8 us かかります（2026 年 10 月に測定。測定方法は crates/net/src/lib.rs に記載しています）。

オリジンのプール

フレーミングをパースすると、オリジン側のソケットを再利用できるかどうかも分かります。Connection: close のない HTTP/1.1 の応答（または keep-alive のある HTTP/1.0 の応答）で、ボディーを完全に読み切ったものは、ソケットをリクエストのきれいな境界に残します。そのようなソケットは 256 個のアイドルなオリジン接続からなる固定のプールに入り、次にプロキシするリクエストは新たに接続する代わりにそこから一つ取ります。プールがなければ、プロキシするリクエストごとに新しいオリジン接続を開き、閉じた接続はそれぞれローカルのポートを TIME_WAIT で保持します。負荷の下では、ループバック上のプロキシはすぐにエフェメラルポートを使い果たし、スループットが崩壊します。
一つのロックで守るプールでは、プロキシするリクエストがすべてそのロックを奪い合うことになります。そこでプール（crates/net/src/pool.rs）はストライプに分けています。16 ソケットのストライプが 16 個あり、各ストライプは専用のキャッシュラインに置いた Mutex<Vec<Arc<TcpStream>>>です。スレッドは最初に使うときにラウンドロビンでホームのストライプを割り当てられ、まずそこにソケットを返したりそこから取ったりします。新たに接続する前には、保持されているロックを待たずに他のストライプも調べます。

```
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()
    })
}
```

プールしたソケットは、アイドルの間にオリジンによって閉じられているかもしれません。プロキシは二つの方法のいずれかでそれに気づきます。書き込みの失敗か、応答の最初のバイトより前のストリームの終端です。どちらの場合もまだ何もクライアントに届いていないので、プロキシはソケットを閉じ、新しい接続で一回だけ再試行します。別のプールしたソケットでは再試行しません。アイドル期間の後ではプールしたソケットがすべて古くなっている可能性があり、二つ目のプールしたソケットを使っていたこの再試行の最初の版は、静かな期間の後の最初のリクエストに毎回 502 で応答していました。再試行が許されるのは、ボディーが完全にバッファーされた安全なメソッド（GET、HEAD、OPTIONS）だけです。応答がないことは、オリジンが変更をコミットしなかった証拠にはなりませんし、分割して中継したボディーを再送することもできません。どちらの場合もクライアントは 502 を受け取ります。

```
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 {
    // 再試行では必ず新しく接続する。アイドル期間の後はプールのどのソケットも
    // 古くなっている可能性があり、二つ目も古ければリクエストが失敗するため。
    let pooled = if attempt == 0 { pool.take() } else { None };
    // ... プールが空なら接続し、アクティビティに結び付けて交換する
    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
                );
            // ... それ以外はすぐに報告する
        }
    }
}
```

単体テストは固定したバイト列を同じ関数に通して中継し、出力が接続ヘッダーを除いてバイト単位で同一であることを確認します（crates/net/src/relay/tests.rs）。デーモンのエンドツーエンドテストは、一つのクライアントソケットでプロキシするリクエストを二つ送り、スタブのオリジンが付けるシーケンスヘッダーによって、二つ目がプールしたオリジン接続を再利用したことを確認します。crates/net/tests/のベクターテストは、記録したプロキシのシナリオ（生のクライアントのリクエストと、用意したオリジンの応答）を再生し、オリジンとクライアントが受け取るすべてのバイトを比較します。

> **オリジンのソケットが Arc である理由** ソケットのディスクリプターは小さな整数で、閉じるとすぐにカーネルが再利用します。シャットダウンは進行中のやり取りを中断しなければなりませんが、その間に別の接続に再利用されたディスクリプターに触れてはなりません。アクティブなディスクリプターの表をロックで守れば防げますが、やり取りのたびにロックのコストがかかります。Sibuna にはそのような表は必要ありません。オリジンのソケットは Arc<TcpStream>です。プールが一つのハンドルを持ち、やり取りが一つを持ち、やり取りの実行中は接続の Activity が複製を持ちます。ディスクリプターは最後のハンドルが破棄されたときに閉じるので、誰かがまだ到達できる間に閉じられたり再利用されたりすることはありません。

アイドル期限

すべてのクライアント接続は、ソケットを IdleTable（crates/net/src/idle.rs）に登録し、Lease を受け取ります。リースの Activity は、接続上でバイトが最後に動いた時刻を保持します。デーモンはリクエストを処理するたびにそれを更新し、プロキシはオリジンからの読み込みごとと、アップロードの 16 KiB ごとに更新し、進行中のやり取りのオリジンのソケットを関連付けます。回収スレッドは最短のタイムアウトの 4 分の 1 ごと（200 ms から 1 秒の間）に起き、タイムアウトより長く無通信の接続を、関連付けたオリジンのソケットとともにすべてシャットダウンします。そのため、どちらの側でブロックしているスレッドもストリームの終端を見て、スロットを解放します。

したがって期限が測るのは無通信の時間であり、応答の長さではありません。動き続けている大きなダウンロードが切られることはありません。アップロードのバイトは信頼できない側から来るので、タイムスタンプを更新するのは 16 KiB ごとだけです。これはクライアントにアイドル期間ごとに少なくとも 16 KiB を求めることになり、最低速度付きのリクエストタイムアウトによる、遅いボディーへの防御です。

この表はスロットごとに小さなロックを一つ持ち、表全体のロックはありません。上で述べた理由から、各クライアントソケットの try_clone と、関連付けた各オリジンソケットの Arc を保持します。Lease は Drop で接続の登録を解除し、回収されたスロットはそのときまでリースが所有し続けます。そのため古い接続が、新しい接続が取ったスロットを解放してしまうことはありません。

アップグレードした接続

検証済みの 101 の後、WebSocket へのアップグレードは双方向に中継します。各方向は一つの固定バッファーを持つ専用のスレッドで動きます。接続自身のスレッドがオリジンからクライアントへのバイトを運び、128 KiB のスタックを持つスコープ付きスレッドがクライア

ントからオリジンへのバイトを運びます。poll(2)を使えば一つのスレッドで両方向を処理できますが、Rust の標準ライブラリはこれを公開していません。ソケットのタイムアウトを持つ二つのスレッドにすれば、中継から unsafe を排除できます。したがってアップグレードした接続は二つのスレッドを持ち、すべての接続がアップグレードされると、接続スレッドの数は--max-connections の 2 倍に達し得ます。

両方向は一つのアイドル期限（--websocket-idle-timeout、既定で 300 秒）を共有します。どちらの方向にバイトが動いても、一つのタイムスタンプが更新されます。ソケットの各待機には残り時間に等しいタイムアウトがあります。タイムアウトした待機は共有のタイムスタンプを読み直し、残りの時間だけ再び待つか、中継を終えます。一方の方向がストリームの終端に達すると、もう一方のソケットの送信側をシャットダウンします（ハーフクローズ）。ある方向が失敗すると、両方のソケットをシャットダウンし、もう一方の方向も終わらせます。アップグレードをプロキシする前に、接続はキャッシュしたポリシーエンジンを手放します。何時間も生き得る接続が古いエンジンを生かし続けないようにするためです。

停止

Running::stop と、同様に Running の Drop は、決まった順序でデーモンを停止します。停止フラグを立て、リスナーに接続してすべての受付ワーカーを起こし、ワーカーを join し、登録されたすべてのクライアントソケットと関連付けたオリジンソケットをシャットダウンし（IdleTable::shutdown）、オリジンのプールを停止し、すべての接続スレッドを join し、回収スレッドを join し、プールしたソケットを閉じます。

SIGINT と SIGTERM は、普通の Rust のコードでこの手順に届きます。デーモンは、スレッドを一つも生成しないうちに両方のシグナルをブロックし（後のスレッドはすべてこのマスクを継承します）、一つのスレッドを sigwait 専用にします。このスレッドは要求を記録し、すぐにメインスレッドを起こします。シグナルハンドラーの中で動くコードはなく、フラグをポーリングするスレッドもありません。シグナルのモジュール（cmd/sibuna/src/signals.rs）にはデーモンで唯一の unsafe コードがあります。std がラップしていない、いくつかの C ライブラリ関数の宣言です。

> **Drop による解放**　明示的な後始末の呼び出しは、準備と逆の順序で書かなければならず、早期リターンのたびにそれらと照らし合わせて読まなければなりません。Sibuna では、各資源が Drop で自分自身を解放します。Daemon と Running はサーバーを停止し、Lease は接続の登録を解除し、Counted は接続数の上限の枠を返し、Finished はタスクのスロットを解放します。コンパイラーは、スコープから出るすべての経路に、それを通り抜けるパニックの巻き戻しも含めて、それぞれの解放を挿入します。そのため解放を忘れる経路はあり得ません。

測定したこと

この表は、Ryzen 9 5950X 一台でのリバースプロキシモードの Shield を示します。デーモンを四つの CPU に固定し、wrk を他の八つの CPU で 2 スレッド、64 接続で動かし、5 秒の実行を 3 回行った中央値です。行は benchmarks/results/product-latest.md から写したもので、完全な記録は第 VIII 部にあります。

表 7 リバースプロキシモードの Shield（benchmarks/results/product-latest.md より）

ワークロード	ステータス	req/s	p50 / p99 (us)	リクエストあたりの CPU (us)
セッションありで許可、プロキシ	200	138,965	438 / 857	28.57
チャレンジ（中間ページ）	200	195,188	162 / 328	13.73
静的な許可、プロキシ	200	145,673	415 / 746	27.27
WAF で拒否（セッションあり）	403	209,507	151 / 306	12.17

wrk は 64 件のリクエストを処理中に保つので、CPU 律速のプロキシの経路ではリクエストが四つの CPU を待って列を作り、そこでの p99 はリクエスト一件のコストではなくスケジューラーの裾です。オープンループの生成器は、一定の速度で送信し、各リクエストを本来の送信時刻から計時するので、同じ負荷を与えたときのレイテンシーを測れます。リバースプロキシモードの許可されたリクエストでは、p99 は毎秒 10,000 リクエストで 65 us、100,000 で 100 us です。与える負荷 150,000 では四つの CPU が飽和し、達成されるのは毎秒 134,344 リクエストで、キューは秒単位まで伸びます（benchmarks/results/latency-latest.md）。温まった keep-alive 接続は、どの判断もヒープ割り当てなしで処理します。cmd/sibuna/tests/steady_state.rs はデーモンの割り当てを数え、crates/net/tests/steady_state.rs は、プロキシした 500 件の keep-alive リクエストで割り当てがゼロであることを確かめます。

演習 5.1. クライアントが 200 KB のアップロードを送ります。どのバイトが 64 KiB のバッファーに置かれ、どのバイトが relay_body によってアップロードバッファーを通して中継されるかをたどり、プロキシが中継の前にヘッダー部をオリジンに書き込まなければならない理由を説明してください。

割り当てを行わない並行状態

この章の学習目標　シャードのロック、Robin Hood 使用済み集合、GCRA リミッター、バージョン付きの禁止テーブル、MPSC インシデントリング、そして Arc と世代を確認するキャッシュによるポリシーエンジンの read-copy-update 方式の公開を分析します。

sibuna-store のどのストアも、構築時に固定の容量を一度だけ確保し、その後はメモリを確保することも、グローバルなロックでブロックすることもありません。バックグラウンドで掃除するスレッドも必要ありません。期限切れの状態はその場で回収されます。

ロックとシャーディング

Sibuna の表のクリティカルセクションは数十命令の長さで、カーネルの futex の往復よりはるかに短いものです。表はキーのハッシュで 16 個のシャードに分かれているので、二つの操作が競合する確率は1/16です。各シャードは専用のキャッシュラインに置かれるので、隣り合うロックが偽共有を起こすこともありません。

表は `std::sync::Mutex` を使います。競合がなければ compare-and-swap 一回、保持者が動いている間は短いスピン、その後 futex で待機します。別の案は、待機ループでスピンループのヒントを使う二状態のアトミックなスピンロックです。SID 0010 は、両者を測定で選ぶよう求めました。スピンロックは競合のない操作あたり約 1 ナノ秒節約しますが、競合の下では負けます。

表 8 Ryzen 9 5950X でのシャードロックのスループット（2026 年 10 月）

ワークロード	Mutex	スピンロック
使用済み集合、16 スレッド	10.8 M/s	8.6 M/s
使用済み集合、64 スレッド（CPU 数より多い）	11.2 M/s	6.9 M/s
六つの集中するアドレスでの GCRA、16 スレッド	44.9 M/s	19.3 M/s
六つの集中するアドレスでの GCRA、64 スレッド	37.4 M/s	6.5 M/s

エッジサーバーは CPU より多くの接続スレッドを動かします。保持者がプリエンプトされたスピンロックでは、待機者全員がタイムスライスを空費するので、ミューテックスを使い続けます。ポイズンされたロックはエラーを伝播させずにそのまま入ります。シャードを保持している間にパニックする操作はなく、仮にパニックしてもシャードは構造的に正しいままです。

Robin Hood 使用済み集合

使用済み集合に入るのは受理された証明だけです（第 III 部）。16 個のシャードを持ち、各シャードは `Option<Entry>` の 4,096 スロットからなるオープンアドレス法の配列です。項目は `{ tag: [u8; 16], expires_at: u64, dist: u8 }` です。Robin Hood 挿入（Celis、1986）は、入ってくるキーよりもホーム位置に近い占有者を押しのけるので、探査長がきつくまとまります。検索は、自分のホーム位置からの距離よりもホーム位置に近い項目に出会った時点で止まります。

実装：`Table::insert` 使用済みのタグを期限とともに挿入し、途中で期限切れのスロットを回収します。`crates/store/src/spent.rs`（クレート `sibuna-store`）にあります。

```
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(());
        };
        // 期限切れの占有者を上書きしてよいのは、新しい距離が小さくない場合だけ。
        // そうすれば、このスロットを越えて探索するキーも早期終了の判定を
        // 引き続き通過する。
        if e.is_expired(now) && carry.dist >= e.dist {
            *e = carry;
            return Ok(());
        }
        if carry.dist > e.dist {
            std::mem::swap(&mut carry, e);
            // 押し出された期限切れのエントリーは単に捨てる。
            if carry.is_expired(now) {
                return Ok(());
            }
        }
        idx = (idx + 1) % SHARD_CAPACITY;
        carry.dist += 1;
    }
    Err(SpendError::Full)
}
```

微妙なのは期限切れのスロットの規則です。期限切れの項目をその場で上書きできるのは、ホーム位置からの距離が少なくとも同じだけあるキーだけです。そうしないと、そのスロットを越えて探査する後のキーが早く止まってしまい、見失われます。押しのけられた期限切れの項目は単に捨てられます。掃除のスレッドは必要ありません。挿入ではまず、何も変更せずに押しのけの経路をシミュレートし（can_insert）、有効なタグを失わずに上限内の押しのけで成功できることを確認します。

> **不変条件**　拒否された挿入が有効なタグを追い出すことはありません。したがって、使用済みで期限切れでないタグは常に見つかり、拒否（SpendError::Full）はサービスを拒否することはあっても、リプレイを可能にすることはありません。

スロットは Option<Entry>です。空のスロットには読むべきタグがないので、読み手が確認を忘れ得る occupied フラグはなく、両方の場合を扱うまで match はコンパイルできません。エラー型も同じく正確です。使用済み集合が失敗する仕方は二つ（DoubleSpend と Full）で、SpendError はちょうどそれらを名指しします。

GCRA：一つの整数によるレート制限

Generic Cell Rate Algorithm（ATM Forum、1996）は、リーキーバケットの仮想スケジューリング形式です。クライアントごとに一つの理論到着時刻（TAT）を保存します。窓Wあたりの正の上限Nに対して、送出間隔を$T = \max(1, \lceil W/N \rceil)$、バースト許容量を$\tau = (N-1)T$とすると、

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

定理（トークンバケットの上限） t_1での最初の適合した到着の後、k番目の適合した到着は$t_k - t_1 \ge (k-1)T - \tau$を満たします。したがって長さ$L$の任意の区間に含まれる適合した到着は高々$N + \lfloor L/T \rfloor$件です。$N$件のバーストの後、$T$ごとに 1 件です。定義したバーストを超えて固定窓の境界による歪みが生じることはなく、リクエストごとの履歴もありません。

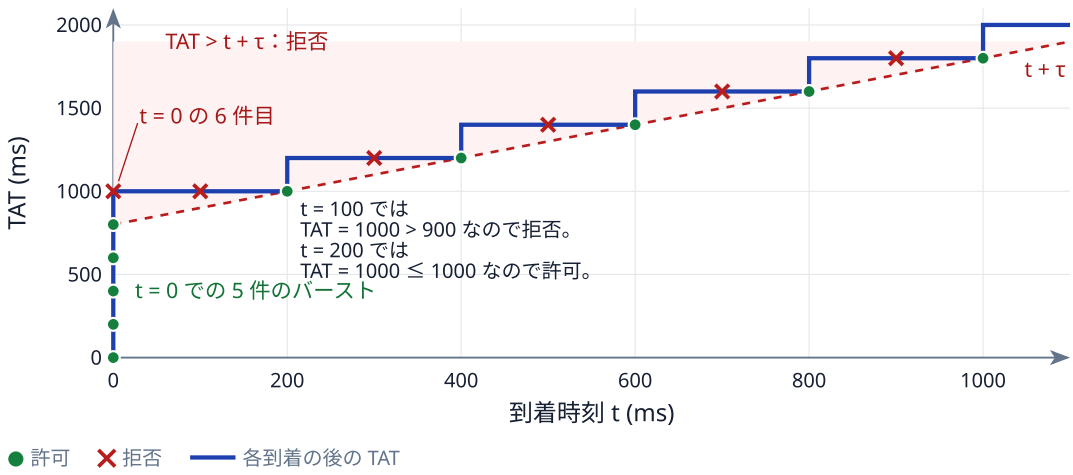


図 7 毎秒 5 件の上限での GCRA による許可：5 件のバーストの後、200 ms ごとに 1 件。到着時の TAT が直線$t + \tau$上かその下にあれば、その到着は許可されます。

例題：毎秒 5 件を手で計算する　$N = 5$、$W = 1000\text{ms}$ とすると、$T = 200\text{ms}$、$\tau = 800\text{ms}$ です。アイドル状態だったクライアントが$t = 0$で 6 件のリクエストを送るとします。1 件目は空のセルを見つけます。$\mathrm{TAT} = \max(0, 0) = 0 \le 800$なので適合し、$\mathrm{TAT} \leftarrow 200$となります。続く 4 件も同じように適合し、$\mathrm{TAT} = 1000$を残します。6 件目は$1000 > 0 + 800$となり、`Retry-After` $\mathrm{TAT} - \tau - t = 200\text{ms}$で拒否されます。$t = 199$でのリクエストもまだ$1000 > 999$となって拒否されます。$t = 200$では$1000 \le 1000$となって適合し、$\mathrm{TAT} = 1200$を残します。`crates/store/src/limiter.rs` の単体テスト `gcra_admits_exactly_the_burst_then_drains_at_the_emission_rate` は、まさにこれらの数値を確かめます。

実装：arrive　ロックしたシャードのセルに到着を一つ適用し、判定、再試行までの遅延、残りの予算を返します。crates/store/src/limiter.rs（クレート sibuna-store）にあります。

```
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)
}
```

シャードの check はロックを取り、ロックしたセルを渡して arrive を呼び出します。ガードは check が戻るときにロックを解放します。セルは 16 バイト（キーと TAT）で、16 スロットの探査窓を持つ 512 スロットのシャードが 16 個、全体で約 128 KiB です。TAT が$t - \tau$より古いセルは空になっており、その場で回収されます。Retry-After ヘッダーも同じ算術で計算します。探査窓が飽和すると、有効なクォータの状態を追い出すのではなく新しいクライアントを拒否し、判定もそのことを示します（capacity_exhausted）。そのため呼び出し側は、飽和による拒否を禁止へと格上げしません。設定したレートがゼロなら、リクエストを拒否します。一つのリミッターが複数のルールに使われます。セルのキーはルールのスコープをシードにしたクライアントアドレスの Wyhash なので、異なるスコープの同じクライアントは独立した予算を持ちます。

バージョン付きの禁止テーブル

禁止には、アトミックなバージョン、キー、期限のフィールドを持つ 4,096 個のスロットを使います。禁止したアドレスの鍵付きハッシュでオープンアドレス法により配置し、探査窓は 8 スロットです。書き手は他の書き手とミューテックスで直列化され、各更新をバージョンの

増加で挟みます。読み手はロックを取らず、安定した偶数のバージョンだけを受け入れ、並行する変更があれば再試行します。

> **実装：BanTable::ban と BanTable::read_slot**　書き手による挟み込みと、読み手による一貫したスナップショットです。crates/store/src/ban_table.rs（クレート sibuna-store）にあります。

```
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)];
    // キーと期限は論理的に一つの値。置き換えを奇数のバージョンで挟み、
    // 読み手が古いキーと次のキーの期限を組み合わせられないようにする。
    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);
        }
    }
}
```

逐次一貫性によって、古い識別子と置き換え後の期限が混ざることを防ぎます。すべてのアクセスが一つの全順序に属するので、議論が単純になります。読み手はミューテックスを取りませんが、書き手を待つことがあり、ウェイトフリーではありません。これらのアトミック操作には実際のコストがあり、無償と仮定するのではなく、HTTP のハーネスで間接的に測定しています。探査窓が満杯なら、期限切れのスロットを、なければ最も早く期限が切れるスロットを上書きします。clear はすべての禁止を解除しますが、キーはその場に残します。空にしたスロットが、探査の連鎖の先にある衝突したキーを隠してしまい得るからです。

MPSC インシデントリング

Shield の運用構成がリクエストを拒否したり、ハニーポットが反応したりすると、インシデントは約 1.3 KB の固定のレコード（クライアントのアドレス、User-Agent、メソッド、パス、カテゴリー、ペイロードで、それぞれ容量で切り詰めます）にコピーされ、上限のある複数生産者・単一消費者のリング（Vyukov のシーケンス番号付きの設計、512 スロット、BoundedQueue<IncidentRecord, 512>）に積まれます。生産者は接続スレッドで、消費者はストレージスレッドです。リングが満杯なら、応答をブロックするのではなく最も新しいレコードを

捨てます。これは永続性の保証ではなく、明示的な損失のポリシーです。破棄のカウンターによって、その損失を観測できます。

```
/// `incident`をリングにコピーする。リングが満杯なら捨てて、その件数を数える。
pub(crate) fn record(&self, incident: &Incident<'_>) {
    if self.queue.push(IncidentRecord::from_incident(incident)).is_err() {
        self.metrics.incidents_dropped.bump();
    }
}
```

ストレージスレッドは、レコードを保留中のバッチに移した後、コミットが確認されるまで、用意した行とともにそれを保持します。データベースのエラーで黙って捨てることはなく、再試行では同一のバッチを書き込みます。
各スロットはシーケンス番号を持ち、それによって生産者はスロットが空いているかを、消費者はスロットが埋まっているかを知ります。そのため、どちらの側もロックを取りません。このようなリングは安全な Rust では書けません。生産者と消費者が一つの配列の異なるスロットに同時に触れ、どのスレッドがどのスロットに触れてよいかを証明するのは型システムではなくシーケンスのプロトコルだけだからです。そのためモジュール crates/store/src/ring.rs は値を UnsafeCell<MaybeUninit<T>>のスロットに保持し、各 unsafe ブロックは、それを正当化するプロトコルの手順を名指しします。

```
if diff == 0 {
    // Relaxed：CASは、どの生産者が位置`pos`を持つかを決めるだけ。
    match self.tail.0.compare_exchange_weak(
        pos, pos.wrapping_add(1), Ordering::Relaxed, Ordering::Relaxed,
    ) {
        Ok(_) => {
            // SAFETY: CASによってこのスレッドが位置`pos`を得ており、
            // `sequence == pos`はスロットが空であることを示す。他の生産者は
            // `pos`を取得できず、消費者は下のReleaseストアより前には
            // これを読まない。
            unsafe { (*slot.value.get()).write(value) };
            // Release：書き込んだ値を消費者に公開する。
            slot.sequence.store(pos.wrapping_add(1), Ordering::Release);
            return Ok(());
        }
        Err(current) => pos = current,
    }
} else if diff < 0 {
    return Err(value);
}
```

リングの二つの約束事は、コメントではなく型になっています。リングが満杯のとき、push は Err(value)を返してレコードを返却し、数えるか捨てるかを呼び出し側に決めさせます。単なる false ではレコードが失われます。リングの消費者は一つで、pop は &mut self を取り、共有されたコードはまず唯一の Consumer を取得しなければなりません。そのため二つ目の消費者はコンパイルできないか、実行時に拒否されます。

read-copy-update によるエンジンの公開

ポリシーが変わると、ポリシーエンジンはホットパスの外で再構築されます。リクエストが作りかけのエンジンを目にしてはならず、リクエストがまだ読んでいる間にエンジンが解放されてもなりません。Sibuna は各エンジンを `EngineSlot` の中の `Arc<Engine>` として公開し、各接続は `EngineCache` を通してそれを読みます。

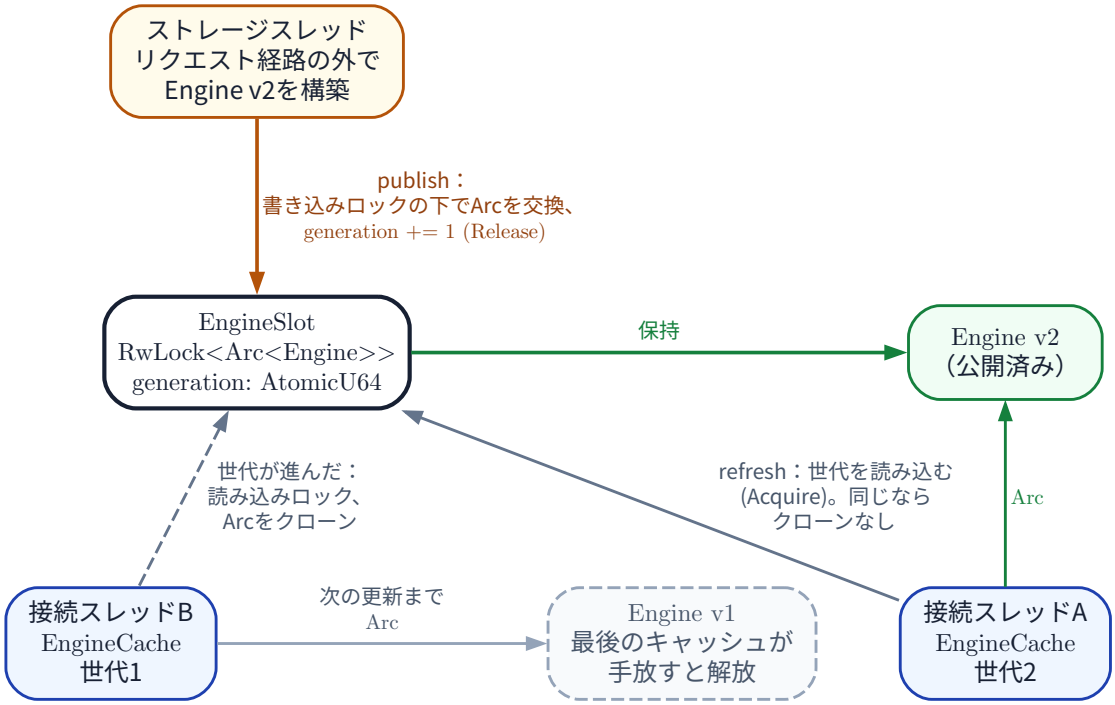


図 8 エンジンの公開。ストレージスレッドは完成したエンジンを公開し、世代を進めます。各接続は一回の acquire ロードで世代を確認し、公開があった後でだけ公開された Arc を複製します。古いエンジンは、それを保持する最後のキャッシュが手放したときに解放されます。

実装：`EngineSlot::publish` と `EngineCache::refresh`　完成したエンジンの公開と、一つの接続から見たその姿です。`cmd/sibuna/src/slot.rs`（クレート `sibuna`）にあります。

```
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);
    // 古いエンジンはロックの外で破棄する（これが最後の保持者だった場合）。
    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 {
        // 上の読み込みとこのクローンの間に公開があると、新しいエンジンに古い世代の
        // ラベルが付く。次の更新で単に読み込み直せばよい。
        self.generation = generation;
        self.engine = None;
    }
    self.engine.get_or_insert_with(|| slot.load())
}
```

世代は、エンジンを配置した後に release の順序付けで格納し、acquire の順序付けでロードします。新しい世代を見た読み手は、その前に格納されたエンジンも見ます。コメントで名指しした競合は無害です。読み手が世代をロードしてから複製するまでの間に公開が起きると、読み手は新しいエンジンを古い世代番号の下でキャッシュし、次の更新で不一致に気づいて再びロードします。読み手は一件のリクエストの間、エンジン一つ分遅れることがありますが、作りかけのエンジンを見ることは決してありません。スロットに格納されるのは完成したエンジンだけだからです。

> **不変条件**　リクエストが読むエンジンはどれも完成しており、リクエストや接続のキャッシュが保持している間は、どのエンジンも解放されません。前者は、`publish` が格納する前にエンジンが完全に構築されるから成り立ちます。後者は、すべての読み手が `Arc` を保持し、エンジンは最後の `Arc` とともにだけ破棄されるから成り立ちます。

なぜ単純なロックではなくキャッシュなのでしょうか。SID 0010 は、16 スレッドで読み込みのコストを測り、`RwLock<Arc<Engine>>` と安全な Rust によるロックフリーの代替案のどちらかを選ぶよう求めました。ロック経由の読み込みは、ロックの語と参照カウントという、すべてのコアが書き込む二つのキャッシュラインに触れます。これに対し世代は、公開のたびに一度書かれるだけで、それ以外は読まれるだけなので、各コアが自分のコピーを保持できます。

表 9 エンジンの読み込み一回のコスト、Ryzen 9 5950X、16 スレッドがループで読み込み（2026 年 10 月）

読み手	1 スレッド	16 スレッド
`RwLock` の読み込みガードのみ	4.2 ns	752 ns
`RwLock` の読み込みと `Arc` の複製	7.6 ns	833 ns

読み手	1 スレッド	16 スレッド
キャッシュした Arc での世代の確認	0.8 ns	1.8 ns

競合の下では、キャッシュによる読み込みはロックより 400 倍以上安価です。そのためリクエストの経路ではキャッシュを使い、RwLock は更新という遅い経路でだけ使います。この測定は cmd/sibuna/src/slot.rs の無視されたテスト read_cost_under_contention です。リクエストの経路では使うたびに更新するので、リクエストはその時点で最新のエンジンを見ます。エンジン自体は小さく、コンパクトなヒープ上の表で 209,052 バイト、約 204 KiB です（benchmarks/results/primitives-latest.json の engine_bytes）。128 個のルール、128 バイトのルール名、8,192 個のトライのノードを収められます。

読み手のカウントではなく Arc を使う理由 固定メモリによる代替案は、エンジンのスロットを二つ持ち、それぞれに読み手のカウントと専用のアロケーションアリーナを持たせます。リクエストはカウントを増やしてスロットのポインターを再確認することでスロットを固定します。読み手のロードと増加の間に書き手が入れ替える競合を閉じるには、両方のアトミック操作が逐次一貫でなければなりません。構築側はポインターを入れ替え、古いスロットの読み手がいなくなるまでスピンしてから、そのスロットのアリーナをリセットして再構築します。したがって、再構築をまたいでスロットを保持する読み手は構築側をブロックします。プロキシの入出力が始まる前にすべてのリクエストがスロットを手放さない限り、遅いオリジンが次の公開を停滞させてしまいます。

Arc の参照カウントでは、このどれも必要ありません。ストレージスレッドはスナップショットごとに新しい Engine を構築して公開します。何も再利用しないので、予備のスロットもアリーナのリセットもスピンもありません。publish が読み手を待つことはありません。古いエンジンは最後の保持者が手放すまで生きるだけで、遅いオリジンが遅らせられるのはその時点だけで、次の公開は遅らせられません。判定は借用を終えるために、一致したルール名を 128 バイトのバッファーにコピーします。判断は接続のキャッシュを通してエンジンを借用しており、応答の経路は接続のコンテキスト全体を可変で必要とするからです。何時間も生き得るアップグレードした接続は、プロキシする前にキャッシュしたエンジンを手放します。

演習 5.2. GCRA の定理を使って、--rate-limit 100 --rate-window 10 のとき、アイドル状態の後の最初の 3 秒間に一つのクライアントが通せるリクエストの最大数を計算してください。10 秒の固定窓のカウンター、および本来のスライディングウィンドウのログと比べてください。

不変条件を説明してください. 期限切れの Robin Hood の項目を上書きできるのが、少なくとも同じだけの距離を持つキーだけである理由を、三つのスロットの例を使って説明してください。

第 VI 部

オートマトン、トライ、意味解析ファイアウォール

1 マイクロ秒未満の分類を支えるアルゴリズムを調べます。タグ付きの Aho-Corasick オートマトン、IPv4/IPv6 の基数トライ、意味解析 WAF のバイトクラスによるトークナイザー、WEIGH によるスコア付けを持つ宣言的ポリシーエンジンです。

一回の走査による複数パターンの照合

この章の学習目標 この章を読み終えると、Aho-Corasick オートマトンの線形時間の保証を導き、密なテーブルの配置とバイトクラスによる大文字小文字の同一視を説明し、SIMD によるリテラル照合エンジンを採用しなかった理由を論じられるようになります。

密な DFA としての Aho-Corasick

Sibuna には二つのリテラルパターンの集合があります。約 40 個のボットのシグネチャと、約 100 個の攻撃のシグネチャです。一つのフィールドをN個のパターンと一つずつ照合すると$O(NM)$です。Aho と Corasick（1975）のオートマトンは、失敗リンク付きのトライを構築した後、フィールドを一度だけ$O(M)$で走査します。Sibuna は構築時に失敗リンクを遷移表に組み込むので、走査は入力 1 バイトあたり表の読み込み一回で、パターン集合に関する分岐はありません。オートマトンは `sibuna-patterns` クレートにあります。

```
/// `state`（行のオフセット）から`byte`で一歩進む。
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
}
```

状態は一つの平坦な表の中の行オフセットとして保存するので、1 ステップは加算一回と読み込み一回です。状態には、すべての一致状態が他のすべての状態の後に来るように番号を

付けます。そのため一致の検出は `first_match` との比較一回で済み、内側のループで二つ目の表を読み込む必要はありません。

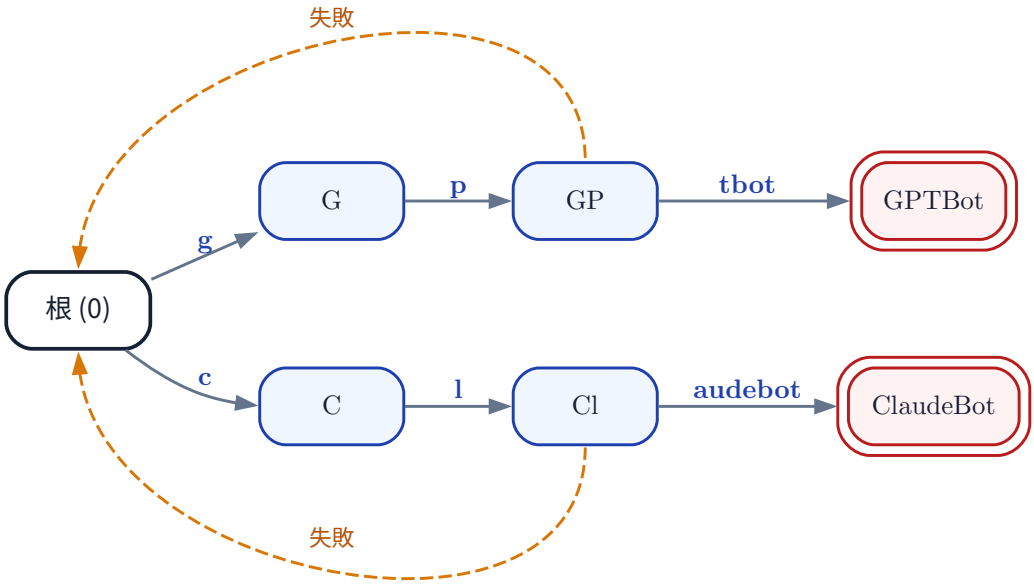


図 9 二つのシグネチャのトライの状態と失敗辺

オートマトンの状態数の予算は実行時のオプションです。ポリシーエンジンは、根一つとすべてのシグネチャの長さの合計を上限として確保します。これはトライの状態数の安全な上限です。これにより、かつての大きすぎる固定サイズの表を避けつつ、すべてのバイトクラスについて密な遷移を保てます。各パターンは小さなカテゴリータグ（0 から 63 なので、タグの集合は 64 ビットの語一つ）を持ちます。大文字小文字の同一視には、オートマトンとともに構築する 256 項目のバイトクラス表を使います。ある文字の大文字と小文字は同じクラスを共有し、どのパターンにも現れないバイトは一つのクラスを共有します。そのため表の 1 行の幅は 256 ではなく、パターン中の異なるバイトの数に 1 を足した値です。エンジンの正確なサイズと、逐次的な部分文字列探索との比較は第 VIII 部に記録しています。後者は同じパターンと、いずれかに一致すればよいという意味論を使います。

正規表現、SIMD エンジン、そして選択

シグネチャの段階には、三つの系統を比較検討しました。

- バックトラックする正規表現（バックトラック型ルールセットの方式）は、リクエストごとに数百のパターンを実行し、一致ごとにメモリを確保します。最も遅く、最も誤検知を起こしやすい選択肢です。
- *SIMD* リテラルエンジン（Hyperscan、Vectorscan）は、シャッフルに基づくニブルマスク（「Teddy」アルゴリズム）で 16 から 32 バイトのブロックを事前に絞り込み、候補をオートマトンで検証します。長い入力と大きなリテラル集合で真価を発揮します。代償は、数メガバイトの C++依存、コンパイル時のパターンデータベース、WebAssembly ターゲットがないこと、そして 100 バイト程度の入力では利点がないことです。フィールド全体の走査は、キャッシュミス一回より短い時間で終わります。

- 密な *Aho-Corasick* と小さな構造トークナイザーの組み合わせは libinjection の系譜で、固定した表で正確に線形時間で動作し、バイナリの外に何も必要としません。

Sibuna のフィールドは短いので、メモリと速度の両面で三つ目の系統が勝ります。第 VIII 部では、8 KiB のボディーで 1 バイトあたり約 1.56 ns を記録しています（意味解析の走査は 12,784.3 ns で、`benchmarks/results/primitives-latest.md` の `8kb_body_semantic_scan` です）。SIMD による事前絞り込みが元を取るのはこのワークロードだけで、SID 0006 はこれを未解決の項目として残しています。

IPv4 と IPv6 の基数トライ

> **この章の学習目標**　128 ビットのトライで最長接頭辞の検索をたどり、IPv4 の根の近道を説明し、トライがクラスターから供給されるレピュテーションの格納場所を兼ねる仕組みを確かめます。

128 ビットのキーに対する一つの二分トライが両方のアドレス族に使われます。IPv6 の接頭辞はそのまま挿入し、IPv4 の接頭辞は `::ffff:0:0/96` に対応付けるので、`/24` は深さ 120 の経路になります。ノードは `u32` で添字付けた一つの平坦な `Vec` に置き、添字 0 は根と「子なし」の両方を表します。そのため検索は高々 128 回の依存した読み込みで、ポインターをたどることはありません。このトライは `sibuna-cidr` クレートの `PrefixTrie` です。

IPv4 の検索のたびに対応付けた接頭辞の 96 段をたどると、IPv4 アドレスの最初のビットに達するまでに 96 回の依存した読み込みがかかります。そこでトライは初期化時に深さ 96 のノードを記録し、IPv4 の検索をそこから始めます。

```
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 アドレスの分類には 32.0 ns、IPv6 アドレスには 79.3 ns かかります（`benchmarks/results/primitives-latest.md` の `ipv4_cidr_classification` と `ipv6_cidr_classification`）。エンジンはトライを二回参照します。終端の `allow`/`deny` の判定はルール表の前に行うので、クラスター全体の禁止はどのルールにも優先します。`challenge` の判定はルール表の後に行います。

意味解析ファイアウォール

> **この章の学習目標**　バイトクラスによる事前走査、タグ付きのシグネチャのオートマトン、恒真式を検出する SQL トークナイザー、HTML とシェルの構造の確認、正規化、そして本物のブラウザーのリクエストを無傷で通さなければならないという誤検知の規律を理解します。

誤検知の教訓

WAF の最初の版は、平坦な部分文字列のリストをすべてのヘッダーと照合していました。リストには/*が含まれており、どのブラウザーも Accept: text/html,...,*/*;q=0.8 を送ります。本物のブラウザーはすべて SQL インジェクションに分類され、403 を受け取りました。この教訓が設計を形作りました。検出器が反応してよいのは構造に対してだけです。テストスイートは現在、select、--、アポストロフィーを含むコメント本文も含めた完全な Chrome のリクエストをエンジンに通し、問題なしと判定されることを確認しています。

一回の走査でのバイトクラス

検査する各フィールドを一度走査し、どのバイトクラスが現れるかを集めます。引用符、=、<、シェルの区切り文字、%、NUL、正規化のきっかけ、連続した空白です。各検出器は再走査せずにこのクラスを参照し、正規化した二回目の走査は、きっかけが見つかったときだけ行います。走査は 1 バイトあたり 256 項目の表の参照一回で、sibuna-policy の WAF モジュールにあります。四つのアキュムレーターで OR の連鎖を短く保ち、二つのバイトの組（/*と、隣り合う二つの空白）は、単独のバイトがまだ正規化を要求していないときだけ探します。

```
pub(crate) fn scan(text: &[u8]) -> Classes {
    let (chunks, remainder) = text.as_chunks::<8>();
    // 四つのアキュムレーターでORの連鎖を短く保ち、ロードを連続して発行できるようにする。
    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)];
    }
    // バイトの組が意味を持つのは、単独のバイトがまだ正規化を要求していない間だけ。
    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 }
}
```

検出器

- シグネチャ。すべてのカテゴリーの強いシグネチャは、一つのタグ付きオートマトンにあります。ヒットは、waf:sqli、waf:xss、waf:path-traversal、waf:rce という名前の終端の deny です。
- *SQL インジェクション。*一回の走査で語を読むスキャナーが、英数字の各語（最大 7 バイト）をキーワード表で引き、or/and では literal = literal という恒真式を確認します。id=1 or 1=1 は終端です。それ以外の場合は引用符のバイトが必須で、証拠のスコアは次のうち 4 以上でなければなりません。引用符（1）、最大二つのキーワード（2）、コメント記号（1）、

= (1) です。admin'--はそれだけで終端です。"it's a group order from the shop"のスコアは 3 で、通過します。

- クロスサイトスクリプティング。<が必須です。スクリプトを載せられるタグが、属性の位置にある on<event>=属性と同時に現れたときに反応します。onboarding=1 は数えません。
- コマンドインジェクション。区切り文字が必須です。;、|、&&、$(、バッククォート、改行の後に既知のコマンド名が続くときに反応します。status ok; done | next は通過します。
- パストラバーサル。シグネチャ（../、..%2f、/etc/passwd、id_rsa など）に加えて、NUL と %00。

文法上正当に引用符やアスタリスクを含む構造的なヘッダー（Accept*、Content-Type、Sec-*、If-*など）は飛ばします。パス、クエリ、User-Agent、その他のヘッダー、ボディーの先頭 8 KB を検査します。パーセントエスケープ、プラス記号、コメントの開始記号、まとめられる空白を含むフィールドは、8 KB のスタック上のバッファーに正規化してもう一度検査します。そのため 1%27/**/UnIoN/**/SeLeCt や %252e%252e/は、元のシグネチャに帰着します。

コスト

検査のコストは、リクエストの数だけでなく、エンジンに渡されるフィールドとバイトによって決まります。ベンチマークの章では、Gate、Shield、8 KB のボディーの場合を別々に報告します。8 KB という上限は検査の制限であり、アップロードの残りが安全だという主張ではありません。forward-auth を使うイングレスは、関係するリクエストの情報を渡さなければなりません。渡されなかったバイトを、このプロセスは検査できません。

宣言的ポリシーエンジン

> **この章の学習目標** ルールのモデル、評価の順序、しきい値付きの WEIGH によるスコア付け、JSON によるポリシーの文法を読みます。

ルールは任意の条件（パスのパターン、User-Agent のパターン、最大四つのヘッダー、最大八つの IPv4/IPv6 CIDR ブロック、一つの JA4 フィンガープリントまたは接頭辞）の論理積で、動作 ALLOW、DENY、CHALLENGE、WEIGH のいずれか、任意のルールごとの難易度（作業ビット）とアルゴリズム、符号付きの weight を持ちます。評価の順序は次のとおりです。

1. 意味解析 WAF（Shield のみ）：違反は終端の deny です。
2. レピュテーショントライ：deny または allow の接頭辞は終端です。
3. ルールを順に：最初の終端の一致で返ります。WEIGH の一致は累積します。
4. スコア：合計が負なら allow。合計が≥ 40なら deny。合計が≥ 10なら challenge で、しきい値を 5 点超えるごとに作業ビットを一つ加え、最大で六つまで加えます。
5. バイパス対象の静的パス、次にトライの challenge の判定、次にボットのオートマトン。
6. 既定の動作。ポリシーファイルで別に指定しない限り challenge です。

組み込みのルールは、/.well-known/*、/favicon.ico、/robots.txt、/__sibuna/*を許可し、CF-Worker のクライアントと Amazonbot を拒否し、Mozilla を含むすべての User-Agent にチャレンジします。ブラウザーの文字列は自由に偽造できるので、それだけで許可すればゲートは飾りになってしまいます。

```
{
  "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 }
  ]
}
```

実行トレースの例：有効なセッションを持つエンコードされた攻撃

有効なセッションを持つクライアントからの `GET /search?q=%2527%25200R%25201%253D1` を考えます。まず HTTP パーサーが、バイトをコピーせずにパスとクエリを分けます。ローカルの制限が実行され、次に Shield がリクエストを検査します。パーセント記号があるので、このフィールドは正規化の対象になります。一回のデコードで `%27%200R%201%3D1` が現れ、二回目で `' OR 1=1` が現れます。

トークナイザーは引用符、キーワード `OR`、等しいリテラルのオペランドを見つけます。拒否はポリシーの終端の結果です。有効なセッションがあっても許可には変わりません。セッションの確認が解決するのはチャレンジの判断だけです。ストレージのフックは上限のあるインシデントレコードをコピーし、SQL は後で実行されます。オリジンへの接続がアイドルの間、応答の経路がエンジンのスナップショットを保持し続ける必要はありません。

演習 6.1. ペイロードを `O'Reilly` に、次に `order=1` に置き換えてください。どちらも単独では SQL インジェクションの根拠にならないのはなぜでしょうか。部分文字列だけで判定する検出器が生みかねない誤検知の例を挙げてください。

演習 6.2. LDAP インジェクション（`)(|(` のようなパターン）の検出器を追加するのに必要なバイトクラス表の項目を書き、既存のどのクラスのビットを再利用できるかを説明してください。

不変条件を説明してください。 `Accept: */*` が誤検知になった理由、新しい SQL 検出器が必要とする構造上の性質、オートマトンとトークナイザーの作業の分担を、レビュー担当者に説明してください。

第ＶＩＩ部

ブラウザーの作業証明エンジン

サーバー自身の検証器のクレートをコンパイルした *WebAssembly* モジュール、*Web Worker* のプロトコル、*JavaScript* による代替の証明器、中間ページを見ていきます。

一つのソース、二つのターゲット

この章の学習目標　この章を読み終えると、ブラウザー用モジュールがサーバー自身の `sibuna-pow` クレートからどのようにビルドされるかを説明し、エクスポートされる関数を挙げ、逐次作業の証明器がタブの中で必要とするメモリを説明できるようになります。

ビルドへの組み込み

ブラウザーのソルバーは、サーバーが検証に使うのとまったく同じソースをコンパイルします。`cmd/sibuna-pow-wasm` は `cdylib` 型の `no_std` ライブラリクレートで、`wasm32-unknown-unknown` 向けにビルドされ、依存するのはワークスペースのクレート `sibuna-pow` と `sibuna-crypto` だけです。

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

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

# Cargo.toml（ワークスペースのルート）：モジュールはサイズ優先でビルドする
[profile.release.package.sibuna-pow-wasm]
opt-level = "z"
debug = false
strip = true
```

デーモンはこのモジュールを埋め込みます。そのビルドスクリプト（`cmd/sibuna/build.rs`）は、入れ子の `cargo rustc -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm` をワークスペースのターゲットディレクトリ（`target/wasm32-unknown-unknown/wasm/`）に対して実行します。入れ子のビルドは `wasm` プロファイルのディレクトリだけをロックするので、外側のビルドを待つことはありません。最終リンクには `--compress-relocations` を渡します。スクリプトは結果が WebAssembly のマジックバイトで始まることを確認し、ビルドの出力ディレクトリにコピーします。WebAssembly ターゲットのないビルドでは、`SIBUNA_POW_WASM=<path>` でビルド済みのモジュールを代わりに埋め込みます。そしてデーモンはそのバイト列をバイナリに含めます。

```
/// ブラウザーのソルバー。`build.rs`が`cmd/sibuna-pow-wasm`からビルドする。
pub const WASM_SOLVER: &[u8] =
```

```
    include_bytes!(concat!(env!("OUT_DIR"), "/sibuna-pow.wasm"));
/// ソルバーのWeb Worker。
pub const WORKER_JS: &str = include_str!("../../../web/worker.js");
```

食い違い得る二つ目の実装は存在しません。モジュールはデーモンから/__sibuna/wasm/sibuna-pow.wasm で、1 時間のキャッシュヘッダー（Cache-Control: public, max-age=3600）付きで配信されます。上のプロファイルでビルドすると、両方の層を含むモジュールは 14,744 バイトです（benchmarks/results/primitives-latest.json の wasm_bytes）。

エクスポート

モジュールは六つの関数をエクスポートし、ワーカースクリプトはそれらを名前で呼び出します。

エクスポート	目的
sibuna_get_buffer_ptr() / sibuna_get_buffer_len()	ワーカーがチャレンジ文字列を書き込む 256 バイトの入力バッファー
sibuna_solve_step(ptr, len, bits, start, max_steps)	Hashcash：start から max_steps 個の nonce を探索します。nonce、またはすべてのビットが 1 の番兵値を返します
sibuna_solve_sha256(ptr, len, bits)	nonce 0 から一回のバッチで行う Hashcash（u32::MAX ステップの sibuna_solve_step）
sibuna_posw_solve(len, depth, challenges)	逐次作業：バッファーに対して証明器を実行し、証明の長さを返します。引数が不正なら 0 を返します
sibuna_posw_proof_ptr()	証明器の作業領域における証明のバイト列の開始位置

証明器の作業領域、つまり保持する上位の段、兄弟のスタック、証明は、約 92 KiB の静的な領域です。アロケーターはリンクされず、モジュールにはインポートがありません。Workspace::new は const fn なので、この静的領域はゼロ初期化されたメモリだけを消費し、モジュール内の初期化済みデータにはなりません。

実装：sibuna_solve_step　Hashcash のエクスポートです。線形メモリからチャレンジを借用し、共有の nonce 探索を一バッチ実行します。cmd/sibuna-pow-wasm/src/lib.rs（クレート sibuna-pow-wasm）にあります。

```
#[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: 呼び出し側は、`challenge_ptr`に`challenge_len`バイトの読み取り可能な
        // 領域があり、呼び出しの間は変化しないことを保証する。
        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)
}
```

エクスポートはこのクレートの外部関数の境界であり、unsafe コードがある唯一の場所です。入力バッファーと証明器の作業領域は ExportCell 内の静的変数です。ExportCell は UnsafeCell のラッパーで、理由を明記したうえで Sync と宣言しています。共有メモリを持たない WebAssembly インスタンスは一つのスレッドで動き、各エクスポートは参照を作り、使い、戻る前に破棄するので、一つのセルへの参照が同時に二つ以上生きることはありません。テストがエクスポートを直接呼び出すホスト上では、テスト専用のミューテックスが呼び出しを直列化します。パニックはメッセージを整形せずにトラップします（unreachable 命令）。これにより、モジュールに整形のコードやインポートが入りません。sibuna-pow の関数は全域的で決してパニックしないので、パニックは構成上到達不能です。

接頭辞の事前ハッシュ

Hashcash の内側のループは、チャレンジとコロンを一度だけ吸収し、nonce ごとに SHA-256 の状態を複製します。そのため各候補のコストは圧縮二回ではなく一回です。探索は sibuna_pow::hashcash::Search で、ブラウザー用モジュールとネイティブの参照ソルバーが共有します。

```
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(); // 吸収済みの接頭辞を値としてコピー
    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
}
```

ハッシュの状態は約 110 バイトの単なる値なので、複製は安価です。逐次作業の証明器も同じ工夫を使います。ステートメントχを base_hasher で一度だけ吸収し、ノードごとに複製します。今では検証器も同様です。ステートメントを一度吸収した状態から、チャレンジされた各葉を導きます。これにより、同じダイジェストのまま検証が圧縮t回分安くなります。

サーバーでは、圧縮関数は実行時に検出した x86-64 の SHA 拡張命令または ARMv8 の SHA-2 命令を使います。ブラウザーにはそのような命令がなく、ソルバーは時間のほぼすべてを移植可能な圧縮関数（crates/crypto/src/sha256/portable.rs）で費やします。この関数は WebAssembly 向けに書かれています。64 ラウンドは展開され、メッセージスケジュールは 16 語の窓に置かれ、16 番目以降の各ラウンドはもう必要のない語を上書きします。8 個の作業変数は、ずらすのではなく、ラウンドごとに名前を付け替えます。

```
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);
    // ... ラウンド16から55
    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);
    }
}

/// スケジュール語W(i)を返す。ラウンド16以降ではその場で計算する。
#[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]
}
```

各 eight_rounds!は変数名を一巡させます。マクロはラウンドごとに変数名を一つずつ回転させて渡すので、一つのラウンドが書き込むのは二つだけで、残りの六つを移動することはありません。ラウンドの添字がリテラルなので、スケジュールの位置はすべて定数になり、ブロック全体が分岐のないコードになって、値は WebAssembly のローカル変数に置かれたままになります。この配置は、ループを展開しない 64 語のスケジュールと比べて、V8 で約 20%、ネイティブで約 14% 速く動きます（Rust 実装で測定、2026 年 10 月）。

ワーカーのプロトコル

この章の学習目標　メッセージのプロトコルを読み、代替手段の階層を理解し、WebAssembly と JavaScript の間の符号付き整数の境界が引き起こした番兵値のバグを確かめます。

ページは `web/worker.js` にメッセージを一つ送り、進捗、代替手段、結果のメッセージを受け取ります。

```
// ページ -> ワーカー
{ challenge: spec.id, algorithm: spec.algorithm, difficulty: spec.difficulty,
    challenges: spec.challenges }

// ワーカー -> ページ
{ 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: '...' }
```

ワーカーはまず WebAssembly を試し、`WebAssembly` が存在しない、ポリシーで遮断されている、インスタンス化に失敗する、のいずれかの場合は純粋な JavaScript に切り替えます。JavaScript の証明器は `sibuna-pow` の Rust の証明器をラベル単位で写したもので、バイト単位で同一の出力を生成します。テストは両方の層について固定した nonce と証明を確認し、SID 0010 は、ソルバーがヘッドレスブラウザーで WebAssembly と JavaScript による代替手段の両方で合格することを要求します。JavaScript の証明器はずっと遅いですが、どのブラウザーでも通過できることを保証します。

符号付きの番兵値　WebAssembly は 64 ビット整数を符号付きの `BigInt` として JavaScript に返します。Hashcash のエクスポートは「バッチを使い切った」ことをすべてのビットが 1 の値（`EXHAUSTED`、`u64::MAX`）で知らせますが、JavaScript はこれを `18446744073709551615n` ではなく `-1n` と見ます。元のワーカーは符号なしの値と比較していたので、ループは最初のバッチの後に偽の nonce で終了しました。修正は `BigInt.asUintN(64, result)` という呼び出し一つで、ワーカーはこれを残しています。

```
// WebAssemblyのi64の結果は符号付きのBigIntとして届くので、u64の
// 「探索し尽くした」番兵値（全ビット1）は-1nとして読める。
const found = BigInt.asUintN(64,
    ex.sibuna_solve_step(ptr, encoded.length, bits, nonce, maxSteps));
if (found !== 18446744073709551615n) return found.toString();
```

この教訓は一般化できます。境界を越えるすべての u64 は正規化しなければなりません。Rust は JavaScript 側でこれを強制できません。エクスポートの型は `u64` と言っており、変換はエンジンの中で起こります。

較正

この章の学習目標　難易度の設定を、ブラウザーエンジンとネイティブのシリコンでの実時間に結び付けます。

一台のノート PC での平均解答時間から難易度を選んではいけません。Hashcash には幾何分布の裾があります。ブラウザーで測った時間には、スケジューリング、温度の状態、起動、コンパイルが含まれます。デプロイの較正では、ブラウザーのバージョン、端末、アルゴリズム、パラメーター、繰り返した解答の分布を記録すべきです。モジュールの取得とコンパイルは、定常状態でのハッシュ計算と分けて扱ってください。

較正計画の例　bを固定し、モジュールをウォームアップし、対象とする端末の種類ごとに、独立に発行したパズルを少なくとも 100 個解きます。所要時間の中央値と高パーセンタイル、失敗、証明のサイズを記録します。$b+1$で繰り返します。期待試行回数は 2 倍になりますが、個々のサンプルが 2 倍になるとは限りません。遅い端末の予算に照らして設定を選び、その後で取得、解答、検証の流れ全体をテストします。

リポジトリにあるネイティブのプリミティブの測定は、スマートフォンやブラウザーでの解答時間を約束する根拠にはなりません。WASM モジュールは同じ算術を実行しても、起動、実行、メモリのコストが異なり得ます。ネイティブではサーバーは SHA-NI でハッシュを計算しますが、ブラウザーは上で述べた移植可能な圧縮関数を実行します。ワーカーのメッセージは、サーバーのリクエストハンドラーにタイマーを加えずにそれらのコストを測れる境界になります。

中間ページ

ページ（`web/challenge.html`、デーモンに埋め込み）は、カード、進み具合を示さない進捗バー、状態の行を持つ一つの HTML ファイルです。自分の場所に対するチャレンジを要求し（`?path=`）、ワーカーを起動し、`fetch` で解答を送信し、成功すると再読み込みします。`fetch` の呼び出しはジッター付きの指数バックオフで再試行し、検証の `4xx` 応答は診断のタイトルとともに表示します。ページの `Content-Security-Policy` は、ソルバーのブロックというただ一つのスクリプトだけを、その SHA-256 ハッシュで許可します。ハッシュはデーモンが埋め込みページから一度だけ計算します。画面外のハニーポットのリンク（`/__sibuna/honeypot`）は、人にもスクリプトを実行するブラウザーにも見えません。すべての `href` をたどる見境のないクローラーは、自ら禁止されます。

演習 7.1. WebAssembly の証明器が、ノート PC で既定の難易度のチャレンジを 20 ms で解き、JavaScript の逐次作業の証明器はその 50 倍遅いとします。そのノート PC と、4 倍遅いスマートフォンでの代替手段による解答時間を見積もり、代替手段を使ったと報告するクライアントに対するポリシーを提案してください（ヒント：ワーカーは、ページが転送できる `fallback` メッセージを送ります）。

不変条件を説明してください． 二つの実装を互いに照らしてテストするよりも、サーバーの検証器のソースからブラウザー用モジュールをコンパイルするほうが強い保証になる理由を説明してください。

第 VIII 部

実験による評価

コミット済みのハーネスで記録した *Sibuna* の測定を示します。プリミティブのレイテンシー、外部の負荷生成器の下で *CPU* とメモリを計上した製品全体、オープンループの生成器による同じ負荷を与えたときのレイテンシー、そして広くデプロイされている作業証明のゲートである *Anubis* との比較です。ライブのダッシュボードを八つ動かしたコンソールの実行で記録を締めくくります。この部のすべての数値は、コミット済みの結果ファイル（各表がその生成時刻とともに示します）か、設計の記録から得たものです。機能の合格と性能の測定は別の問いに答えるもので、一つのホストでの測定は別のホストでの順位ではありません。

方法

この章の学習目標　この章を読み終えると、この部のすべての数値を再現し、各ハーネスが何を含み何を除くかを述べ、ばらつきの列の意味を説明し、測定した数値と、記録された数値や公表された数値とを見分けられるようになります。

ハーネスは一つの Rust のツール `tools/bench`（バイナリ `sibuna-bench`）のコマンドで、`cargo xtask bench` を通して実行します。別のツールがコンソールの実行を記録します。

表 11 ハーネスとその結果ファイル（すべて `benchmarks/results/` 配下）

ハーネス	測定するもの	結果ファイル
`cargo xtask bench primitives`	一つの CPU での 16 個のプリミティブのワークロードと、ハッシュ化したレートリミッターの経路の 2 行。7 バッチ、操作あたりの中央値と最小・最大のばらつき	`primitives-latest.json`, `primitives-latest.md`
`cargo xtask bench product`	`wrk` の下でのデーモン全体。forward-auth モードとリバースプロキシモードでの Gate と Shield のスループット、レイテンシーのパーセンタイル、リクエストあたりの CPU 時間、ピークの常駐メモリ	`product-latest.json`, `product-latest.md`
`cargo xtask bench latency`	オープンループの生成器による、一定の負荷レートでの許可されたリクエストのレイテンシーのパーセンタイル	`latency-latest.json`, `latency-latest.md`
`cargo xtask bench product --anubis`	Anubis をさらに一つの製品として加えた製品全体のワークロード。`caddy respond` の背後	`comparison-latest.json`, `comparison-latest.md`

ハーネス	測定するもの	結果ファイル
`cargo xtask bench admission --anubis`	forward-auth モードでの Sibuna と Anubis の毎秒の許可操作数。二つの keep-alive クライアントから	`admission-latest.json`, `admission-latest.md`
`sibuna-browser dashboards`（`tools/browser`）	コンソールの機能の実行。生成器がトラフィックを送る間、ヘッドレス Chromium で八つのライブのダッシュボードを動かす	`console-8-dashboards-2026-10-15.md`

`.md` ファイルは要約です。`.json` ファイルは、すべての繰り返し、マシン、リビジョン、測定したすべてのバイナリのパス、サイズ、SHA-256 を保持します。履歴は Git が保持するので、ディレクトリには最新のファイルだけを置きます。一部のファイルは以前の版のハーネスで記録したもので、同じ実行で測定した二つ目のデーモンのビルドの行も含みます。この部では Sibuna の行を報告します。

プリミティブ

各ワークロードは、計時しない最大 1,000 操作のウォームアップの後、計時する 7 バッチで実行します。可変の状態（使用済み集合、レートリミッター）は、各バッチの前にタイマーの外で作り直します。タイマーは個々の操作ではなくバッチ全体を囲みます。操作ごとに時計を読むと、測定する 100 ns 未満の作業と同じだけのコストがかかるからです。そのため操作ごとのパーセンタイルは報告せず、ばらつきはバッチ間のものです。操作あたりの中央値のバッチを、最速と最遅のバッチと並べて報告します。

スイートは一つの CPU に固定した子プロセス（`sibuna-bench primitives-run`）で実行します。共有ホストでは、忙しい SMT の兄弟スレッドや低いクロックがスイートの実行全体を遅くするので、ツールはスイートを 3 回実行し（`--rounds`）、行ごとに中央値が最も小さいラウンドを残します。ラウンドの中の統計は上のとおりです。

製品全体

各モード（forward auth、リバースプロキシ）と各運用構成（Gate、Shield）について、ハーネスはデーモンをメモリ上で`--algorithm hashcash --difficulty 8 --rate-limit 100000000 --idle-timeout 5` と四つのワーカー（その CPU セットの CPU ごとに一つ）で起動します。ブラウザーと同じように、`sibuna-pow` でデーモンの Hashcash チャレンジを解いてセッションを得ます。その後 `wrk` で、keep-alive 接続を使って四つのワークロードを実行します。2 スレッド、64 接続、5 秒、3 回の繰り返しです。

- `admitted`：有効なセッション Cookie 付きの `GET /private`。リバースプロキシモードではリクエストがオリジンに届き、その応答を中継します。forward-auth モードではデーモン自身が `200` で応答します。
- `challenged`：Cookie のない同じリクエストを、ブラウザーの User-Agent から送ります。デーモンはチャレンジを返します。リバースプロキシモードでは HTML 付きの `200`、forward-auth モードでは `401` です。
- `allowed_static`：`curl/8.0` からの `GET /robots.txt`。どちらの運用構成もチャレンジなしで許可します。

- attack：有効なセッションに q=' OR 1=1--を加えたもの。拒否するのは Shield だけです（403）。Gate は許可し、リバースプロキシモードではオリジンに中継します。

オリジンはスタブ（sibuna-bench origin）です。標準ライブラリだけを使い、接続ごとに一つのスレッド、keep-alive、41 バイトのページです。リクエストあたりの CPU 時間は、実行中の/proc/<pid>/stat から得たデーモンのユーザー時間とシステム時間（10 ms のティック）を、wrk が数えたリクエスト数で割ったものです。オリジンの CPU は含みません。ピークメモリは/proc/<pid>/status の VmHWM で、各実行の前に/proc/<pid>/clear_refs でリセットするので、各実行は自身のピークを報告します。どの数値も繰り返しの中央値で、最小値と最大値は JSON ファイルに並べて残しています。

wrk はクローズドループの生成器で、64 件のリクエストを処理中に保ちます。そのため速いデーモンほど多くのリクエストを受け取り、wrk のパーセンタイルは、遅い応答を待つ間に送らなかったリクエストを除外します（coordinated omission）。その p99 が示すのは、デーモン自身の飽和の下での振る舞いであり、ある負荷でのレイテンシーではありません。その問いにはレイテンシーのハーネスが答えます。

同じ負荷でのレイテンシー

オープンループの生成器（sibuna-bench generate）は、接続ごとに一つのブロッキングスレッドを動かします。64 個の接続はそれぞれ、セッション付きの GET /private を決まった予定で送ります。接続のリクエストkの予定時刻は$t_0 + o + kT$で、oは接続のオフセット、Tはその間隔であり、それ以前の応答とは独立です。次のリクエストの予定時刻を過ぎてから応答が届いた場合、wrk2 と同じく、そのリクエストはすぐに送られ、レイテンシーは予定時刻から数えます。予定時刻に接続がアイドルだった場合、レイテンシーは実際の送信時刻から数えるので、生成器自身のスリープの行き過ぎがデーモンに課されることはありません。

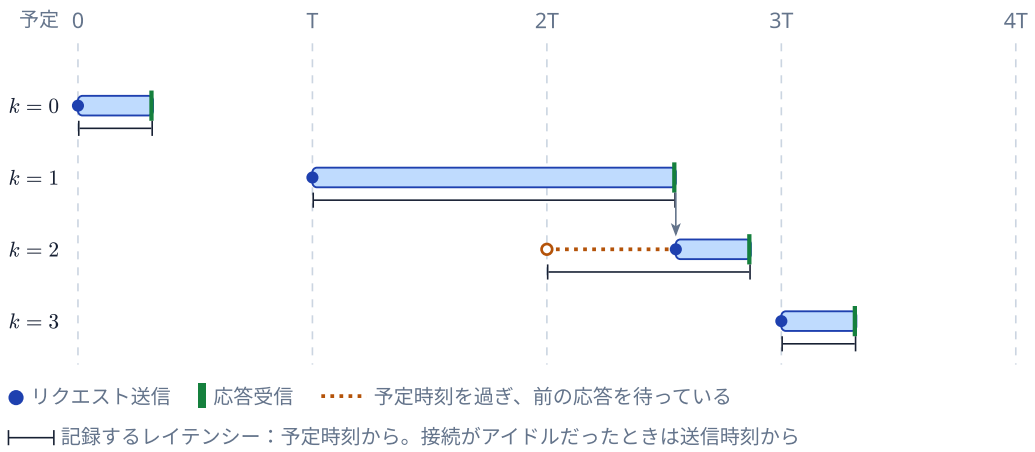


図 10 オープンループの生成器の一つの接続。リクエスト 1 はリクエスト 2 の予定時刻の後に応答されるので、リクエスト 2 はすぐに送られ、そのレイテンシーは 2T から数えます。リクエスト 3 は接続がアイドルなので、送信時刻から数えます。

ハーネスは、Shield の運用構成と許可のワークロードを五つの負荷レートで実行します。毎秒 10,000、25,000、50,000、100,000、150,000 リクエストで、それぞれ 2 秒のウォームアップ、10 秒の測定、3 回の実行です。ハーネスは一回の実行で二つのデーモンのビルドを測定でき

ます。その場合、両方を同じ CPU セットで起動し、レートと繰り返しのたびに交互に切り替え、アイドルの側は CPU を使いません。これにより、マシンのゆっくりした変動が両方に同じように影響します。記録した実行はそのようにしました。パーセンタイルは一回の実行で測定したすべてのリクエストに対する最近順位法によるもので、各数値は 3 回の実行の中央値で、p99 の最小・最大を並べています。

CPU セット

CPU セットは他のどの設定よりも重要です。デーモンは CPU 0 から 3 で動きます（`--cpus`。`--workers` はその集合の大きさです）。`wrk`、レイテンシーの生成器、オリジンは CPU 8 から 15 で動きます（`--load-cpus`）。プリミティブのスイートは CPU 4 で動きます（`--cpu`）。どのプロセスも開始前に `taskset -c` で固定するので、プロセスが作るすべてのスレッドがその集合を継承します。許可のハーネスは、製品に二つの CPU（0 と 1）を、クライアントに CPU 8 から 15 を使います。



図 11 AMD Ryzen 9 5950X での CPU セット。CPU 0 から 7 は一つ目の CCD の 8 コア、8 から 15 は二つ目の CCD の 8 コア、16 から 31 はそれらの SMT の兄弟です。デーモンと負荷生成器がコアを共有することはありません。

他のマシンでは、コアを共有しない集合を選びます（`lscpu -e` でトポロジーが分かります）。

マシンと条件

ファイルは `paxos-zig` で記録しました。AMD Ryzen 9 5950X 16-Core Processor（論理 CPU 32 個）、Linux 7.0.0-28-generic x86_64、ブーストを有効にした `powersave` ガバナー、rustc 1.99.0 です。プリミティブ、製品、レイテンシーのファイルは 2026 年 10 月 15 日に、Anubis との比較は 2026 年 10 月 16 日に記録しました。デーモンは `cargo build --release -p sibuna`、つまり SQLite ストレージとコンソールを組み込んだ既定のビルドです（SID 0010）。製品とレイテンシーのファイルは 49,113,856 バイトのバイナリを示しています。各ファイルはリビジョン（プリミ

ティブは efce14d、製品とレイテンシーの実行は 249d913）を dirty: true とともに記録しています。後者の作業ツリーには、SID 0002 が記録する接続スレッドの再利用がすでに入っており、SID 0002 はその効果について latency-latest.md を引用しています。比較と許可のファイルはリビジョン 54e4ec5 を dirty: true とともに示しています。作業ツリーに tools/bench のハーネスの追加があったからです。そのデーモンはそのリビジョンのリリースビルド（37,450,232 バイト）です。1 分間のロードアベレージは、比較の実行の前に 3.53、許可の実行の終了時に 0.56 でした。
ファイルに記録された 1 分間のロードアベレージは、プリミティブの実行時に 2.46、レイテンシーの実行の前に 1.30、後に 52.31、製品の実行の前に 52.31、後に 33.39 です。高い数値はレイテンシーの生成器の 64 スレッドによるもので、製品の実行はその直後に行いました。

> **測定の範囲**　出力するのはローカルの測定だけです。どの実行も一つのホストでループバックを使います。TLS も WAN も本物のブラウザーもありません。デーモンはストレージなしでメモリ上で動きます。タイマーはバッチを囲み、状態のリセットとウォームアップは計時しません。ハーネスは割り当てを数えません（ファイルには"not instrumented; API/source audit only"とあります）。リクエストの経路で割り当てがないことは、代わりに第 V 部の定常状態のテストで確認します。どのハーネスもデーモンにリンクせず、リクエストの経路にフックを加えません。製品とレイテンシーのハーネスは、SHA-256 で識別した無変更のリリースバイナリを実行します。このホストでの順位は、別のホストでの順位ではありません。

再現の方法

Linux、taskset（util-linux）、PATH 上の wrk が必要です。

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

cargo xtask bench はまず cargo build --release -p sibuna でデーモンをビルドし、次にリリースモードで sibuna-bench を実行します。ツールは自身のバイナリの隣にあるデーモンを見つけます。--rust <path>で上書きできます。cargo xtask bench help はオプションを一覧表示します。その中には--cpus、--load-cpus、--cpu、--rounds、--repetitions、--seconds、--rates、--warmup があります。Anubis との比較を記録したコマンドは、後の Anubis の節に示します。

プリミティブのレイテンシー

表はプリミティブの実行のすべての行を示します。JSON ファイルは各行をサブシステムとワークロードで名付けています（たとえば pow_verify と hashcash_16_bits）。

表 12 Ryzen 9 5950X の 1 コアでの操作あたりのプリミティブの所要時間、ナノ秒（benchmarks/results/primitives-latest.md、2026-10-15T22:34:59Z 生成）

ワークロード	ns/op	最小-最大
Hashcash の検証（16 ビット）	70.2	69.9-71.0
PoSW の検証（深さ 13、t = 16）	22272.4	22257.0-22316.4
ボットのシグネチャ（40 個、Aho-Corasick）	55.7	55.3-57.2
ボットのシグネチャ（40 個、逐次探索）	763.0	762.2-763.5
IPv4 CIDR の検索	32.0	31.8-32.2
IPv6 CIDR の検索	79.3	78.5-79.6
セッショントークン（鍵付き BLAKE3）	155.8	155.3-156.4
セッショントークン（Ed25519）	40587.5	40544.6-40631.1
使用済み集合（Robin Hood）、使用と検索	21.0	20.8-21.3
レートリミッター（GCRA）、クライアントのバイト列	12.8	12.7-12.8
レートリミッター、四つのルールのスコープ、クライアントのバイト列	13.1	13.0-13.2
レートリミッター（GCRA）、ハッシュ化したクライアント	6.2	6.1-6.3
レートリミッター、四つのルールのスコープ、ハッシュ化したクライアント	6.3	6.2-6.4
HTTP のパースと Cookie	132.5	132.3-133.2
分類、Shield のプロファイル	790.8	787.2-793.1
分類、Gate のプロファイル	75.9	75.6-76.1
WAF のボディーの走査（8 KiB）	12784.3	12771.5-12809.2
WAF のボディーの走査（8 KiB）、SQLインジェクションを監査	15485.4	15460.9-15530.1

ばらつきは小さく、どの行のどのバッチも、その行の中央値から 2.8% 以上離れていません。図は同じ行を対数目盛で示すので、6 ns の行と 40 us の行を並べても読み取れます。

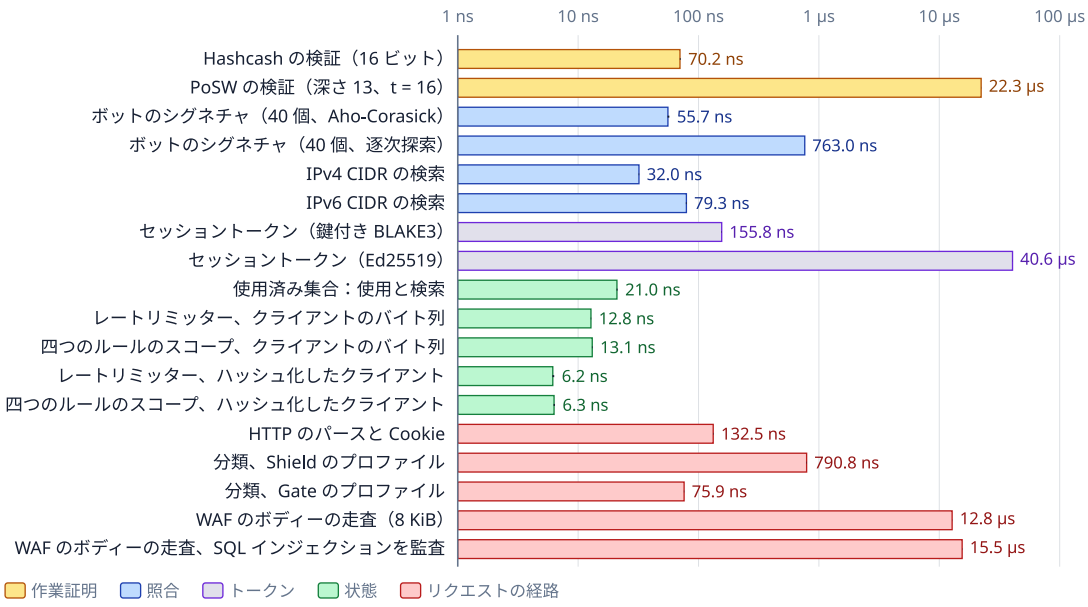


図 12 対数目盛で示したプリミティブの所要時間。benchmarks/results/primitives-latest.json から描いています。

作業証明の検証

Hashcash の検証は、チャレンジ、区切り文字、十進数の nonce をハッシュします。70.2 ns です。逐次作業の検証器はグラフの開示を確認します。22,272.4 ns です。入力の長さが異なるので、ハッシュの呼び出し回数と SHA-256 の圧縮回数は Hashcash とは異なります。どちらの行も SHA-256 に律速され、`sibuna-crypto` は、CPU に SHA-NI 命令があれば実行時に検出して使います（SID 0010）。

照合

密な Aho-Corasick オートマトンと逐次的な部分文字列探索は、同一のパターンと、いずれかに一致すればよいという意味論を使います。同じ 40 個のシグネチャで、オートマトンは逐次探索より 13.7 倍速く動きます（55.7 ns 対 763.0 ns）。その表は状態ごとに 256 列ではなくバイトクラスを使うので（SID 0006、Part F）、ボットのシグネチャの表は 42,244 バイトに収まります（後のフットプリントの表）。CIDR の分類器は 32.0 ns（IPv4）と 79.3 ns（IPv6）です。

トークン

BLAKE3 の MAC トークンは 155.8 ns で検証できます。`sibuna-crypto` は BLAKE3 を移植可能な形で実装しており、SIMD の経路はありません。SIMD の経路は大きな入力で有利ですが、Sibuna が BLAKE3 でハッシュするのは 100 バイト未満の入力で、そこでは移植可能なコードで十分速いからです。Ed25519 のコンパクトなトークンの検証は 40,587.5 ns です。検証器は

幅 5 と幅 8 の NAF の桁を使い、デコードした公開鍵を保持します。Ed25519 の検証一回は、MAC の検証約 260 回分のコストです。MAC が Sibuna の既定のトークン方式です。

状態

使用済み集合の行は、新しいタグを挿入して再び検索します。21.0 ns です。この行の表は 4,096 スロットのシャードが 16 個（65,536 タグ。デーモンの既定は 1,048,576）で、各バッチの前にタイマーの外でその場で空にします。したがって、どのバッチも常駐しているメモリを探査します。以前のスイートは、バッチごとにデーモンの既定サイズのゼロ初期化した新しい表を作っていました。それでは、最初の書き込みでカーネルが表のページを割り当てる時間を計っていました。これは稼働中のデーモンが一度だけ払うコストで、報告すべき行ではありません。

レートリミッターの行には注意が必要です。クライアントのバイト列の二つの行（`gcra_check` と `gcra_four_rule_scopes`）は、バイト列の API を呼び出します。各確認はクライアントのアドレスをバイト列として受け取り、まず OS の乱数から引いた秘密の下で、その鍵付き SipHash-1-3 を計算します。秘密鍵があるので、誰も被害者の探査窓を共有するアドレスを計算できません（SID 0010 と `sibuna-store` の文書）。これらの行は 12.8 ns と 13.1 ns です。

デーモンはリクエストの経路でバイト列の API を使いません。リクエストごとに一度クライアントをハッシュし（`ClientHasher`）、参照するすべての表にそのハッシュを渡します。ハッシュ化したクライアントの二つの行（`gcra_check_hashed` と `gcra_four_rule_scopes_hashed`）は、ループの前にハッシュを計算したうえで、その経路を測ります。確認あたり 6.2 ns と 6.3 ns です。バイト列とハッシュ化の行の中央値の差 6.6 ns は、これらの入力での鍵付きハッシュのおよその代価で、デーモンはこれを表ごとではなくリクエストごとに一度払います。

HTTP パーサー

パーサーの行は、400 バイトのブラウザーのリクエストをパースし、Cookie を一つ検索します。132.5 ns です。クレート自身の計時テストも一致します（`crates/http`：同じマシンで約 135 から 180 ns）。パーサーはゼロコピーで、どのフィールドも接続のバッファーのスライスです。

ポリシーエンジン

ブラウザーのリクエストの完全な分類には 790.8 ns、Gate のプロファイルでは 75.9 ns かかります。これらの行では、リクエストのパース済みのクライアントアドレスを空にしているので、エンジンは評価の中でクライアントアドレスのテキストをパースします。デーモンは評価の前にリクエストごとに一度アドレスをパースするので、これらの行には、デーモンがエンジンの外で払う小さなコストが含まれています。エンジンのヒープ上の表は合計 209,052 バイトです（SID 0002）。

WAF の検査

8 KiB のボディーの意味解析の走査には 12,784.3 ns、つまり 1 バイトあたり 12784.3/8192 ≈ 1.56 ns かかります。二つ目の行は、SQL インジェクションのカテゴリーを監査、その他を強制にして同じボディーを走査します。15,485.4 ns です。いずれかのカテゴリーが強制ではなく監査のとき、各フィールドはカテゴリーごとに検査されます（SID 0004、"inspection modes"）。エンジンはシグネチャのオートマトンでフィールドを一度走査して、シグネチャが現れるカテゴリーの集合を求め、その中から必要なカテゴリーだけを探します。そのため、一つのカテ

ゴリーを監査するコストは、すべてを強制する場合の約 5 分の 1 増しで、カテゴリーごとに走査するわけではありません。

製品全体の比較

> **この章の学習目標** 完全なプロセスとしてのデーモンのスループット、レイテンシー、CPU 時間、メモリを読み、運用構成とモードを比べ、どの行でデーモンが、どの行で負荷生成器が限界を決めているかを述べます。

表は、各運用構成とワークロードについて、5 秒の `wrk` の実行 3 回の中央値を示します。各ワークロードの横のステータスは、ハーネスが期待し、実際に受け取った応答です。

forward-auth モード

イングレスがデーモンにリクエストを許可するかどうかを問い合わせます。オリジンもボディーの中継もないので、これが許可のコストの最も純粋な尺度です。

表 13 forward-auth モード、Gate と Shield。レイテンシーはマイクロ秒（`benchmarks/results/product-latest.md`、2026-10-15T23:15:24Z 生成）

運用構成	ワークロード（ステータス）	req/s	p50	p90	p99	CPU us/req	ピーク 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
Shield	`attack` (403)	210,549	152	175	307	12.32	16192

forward-auth モードでは、デーモンはボトルネックではありません。稼働中のコア数（実時間に対する CPU 時間、JSON ファイルの `cores_busy_median`）は、デーモンの四つの CPU のうち 2.47 から 2.72 にとどまり、どの行も毎秒 197,532 から 211,636 リクエストの間にあります。これは負荷生成器が上限であることを示しています。その上限の近くでは、行の間の数パーセントの差は実行ごとに入れ替わります。Gate の `admitted` の行だけでも、3 回の実行で毎秒 205,151 から 211,513 リクエストの幅があります。このモードでより有用な数値は、リクエストあたりの CPU 時間です。11.73 から 13.77 us で、Gate より Shield のほうが少し多く、チャレンジの応答を描画する `challenged` の行が最も多くなります。

リバースプロキシモード

デーモンはスタブのオリジンの前に置かれ、許可したリクエストと応答を中継します。そのためプロキシする行には、オリジン自身のコストと、ループバックの 1 ホップ分が加わります。

表 14 リバースプロキシモード、Gate と Shield。レイテンシーはマイクロ秒（benchmarks/results/product-latest.md、2026-10-15T23:15:24Z 生成）

運用構成	ワークロード（ステータス）	req/s	p50	p90	p99	CPU us/req	ピーク 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

ここではプロキシする行（admitted、allowed_static、そして Gate が中継する Gate の attack）は CPU 律速で、稼働中のコア数は四つのうち 3.96 から 3.98 です。毎秒 138,965 から 147,303リクエストを、リクエストあたり 26.88 から 28.57 us の CPU で処理します。これはデーモン自身が返す応答の約 2 倍のコストです。challenged の行と Shield の attack の行はデーモン自身が応答するので、forward-auth の行と似ています。飽和の下では wrk の p99 はデーモン自身の速度でのスケジューラーの裾なので、代わりに次の章の裾を読みます。

リバースプロキシの三つの行で、wrk は、想定していなかった 2xx と 3xx 以外の応答を数えました。3 回の実行の合計で、2,162,579 リクエスト中 3 件（Gate の attack）、2,082,640 中 1 件（Shield の admitted）、2,204,358 中 1 件（Shield の allowed_static）です。ファイルには、それがどのステータスだったかは記録されていません。デーモンのメトリクスを付けて実行を再現すると、sibuna_overloaded_total に数えられた 503 の応答が見つかりました。wrk は 64 個の接続を一つのアドレスから開き、これは既定の--max-connections-per-ip とちょうど同じです。接続が 4,096 番目のリクエストの後で閉じられたとき、古い接続が枠を返す前に wrk が再接続していました。現在のデーモンは、閉じる応答を書く前に枠を返します（SID 0002、"bug fix"）。表はこの修正より前のものです。

フットプリント

表 15 デーモンのサイズと常駐メモリ（benchmarks/results/product-latest.json、2026-10-15T23:15:24Z 生成、および primitives-latest.json、2026-10-15T22:34:59Z 生成）

項目	値
デーモンのバイナリ	49,113,856 バイト

項目	値
アイドル時の常駐メモリ、製品の実行	5,592 から 5,788 KiB
ワークロード中のピークの常駐メモリ	15,896 から 16,348 KiB
ワークロード後の常駐メモリ	16,176 から 16,348 KiB
アイドル時の常駐メモリ、プリミティブの実行	6,052 KiB
ポリシーエンジンの表	209,052 バイト（ヒープ）
ボットのシグネチャの表	42,244 バイト（ヒープ）
WAF の表	164,908 バイト（ヒープ）
ブラウザーのソルバーのモジュール（WebAssembly）	14,744 バイト

範囲は四つの製品の実行（二つのモード、二つの運用構成）にわたるもので、ピークについてはその 16 個のワークロードにわたるものです。表のサイズはエンジンが報告するヒープのサイズです。記録にはバイナリのサイズの内訳はありません。

同じ負荷を与えたときのレイテンシー

> **この章の学習目標**　一定の負荷を与えたときのレイテンシーの分布を読み、裾のパーセンタイルをそのばらつきとともに読み、一つの裾の欠陥を測定から修正までたどります。

どちらの表も `cargo xtask bench latency` の一回の実行から得たものです。エラーの列は、生成器の接続、入出力、フレーミングのエラーを数えます。

表 16 forward-auth モード、Shield、一定の負荷レートでの許可されたリクエスト。レイテンシーはマイクロ秒（`benchmarks/results/latency-latest.md`、2026-10-15T23:07:14Z 生成）

負荷 req/s	達成 req/s	p50	p90	p99（最小-最大）	p99.9	max	エラー	CPU us/req
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
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

表 17 リバースプロキシモード、Shield、一定の負荷レートでの許可されたリクエスト。レイテンシーはマイクロ秒（`benchmarks/results/latency-latest.md`、2026-10-15T23:07:14Z 生成）

負荷 req/s	達成 req/s	p50	p90	p99（最小-最大）	p99.9	max	エラー	CPU us/req
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 モードでは、p50 はどのレートでも 24 から 30 us にとどまります。p99 は毎秒 150,000 リクエストまで 51 us 以下、p99.9 は 74 us 以下にとどまります。デーモンはリクエストあたり 13.48 から 16.92 us の CPU を費やします。

リバースプロキシモードでは、毎秒 100,000 リクエストまで p50 は 47 から 56 us で、100,000 での p99 は 100 us、p99.9 は 150 us です。毎秒 150,000 リクエストではデーモンが飽和します。毎秒 134,344 リクエストを完了し、レイテンシーは秒単位まで伸びます。処理能力を超えたオープンループのテストでは、こうなるのが当然です。このレートは、前章のクローズドループでのプロキシのスループット（Shield の `admitted` で毎秒 138,965 リクエスト）に近い値です。その行の四つのエラーは過負荷の中で起きました。

飽和の手前では p99 のばらつきは小さく、最も大きいのは毎秒 50,000 リクエストでのリバースプロキシの p99 で、58 から 66 us です。max の列は、一回の実行での最大 150 万件のリクエストのうちの 1 件です。共有ホストでは、これをデーモンの性質ではなく、ホストの最悪の瞬間として読みます。

測定が変えたこと

外部からの負荷は Sibuna の設計を変えました。デーモンは各接続を専用のスレッドで処理します（SID 0002、SID 0010）。最初のレイテンシーの実行では、`wrk` が示さなかった裾が見つかりました。64 接続で、デーモンが 4,096 リクエストごとに各接続を閉じる場合、許可されたリクエストが毎秒 100,000 件のときの 99.9 パーセンタイルは 0.3 から 2 ms でした。

SID 0002 と `cmd/sibuna/src/tasks.rs` の文書に記録した診断は、スレッドの入れ替わりです。スレッドの生成と終了では、シグナルスタックのマップとアンマップ、スタックのページの返却が起き、そのアドレス空間の変更がデーモンの他のスレッドを停滞させていました。一つの実験で決着しました。接続を一度も閉じない、つまり実行中にスレッドを一つも生成しないビルドでは、同じレートで 54 から 60 us になったのです。修正では接続スレッドを再利用します。接続を終えたスレッドは待機して次の接続を引き受けるので、新しい接続が絶えず来てもスレッドは生成されません。10 秒待機したスレッドは終了し、次の起動か、次にタスクを終えたスレッドがそれを join します。そのため、バーストの後にスタックがメモリに残り続けることはありません。接続がパニックしたスレッドは、待機せずに終了します。同時に処理する接続は、以前と同じく最大`--max-connections` です。

上の表は修正後の実行のものです。毎秒 100,000 リクエストでの p99.9 は、forward-auth モードで 62 us、リバースプロキシモードで 150 us です。修正前の実行は結果ファイルとして残していないので、その数値は SID 0002 とこの節にだけ現れます。

例題：停滞が p99.9 に現れた理由 64 接続で毎秒 100,000 リクエストのとき、各接続は毎秒100000/64 = 1562.5リクエストを運びます。デーモンは keep-alive 接続を 4,096 リクエストの後に閉じるので（`cmd/sibuna/src/connection.rs` の `MAX_REQUESTS_PER_CONNECTION`）、接続は4096/1562.5 ≈ 2.62秒続き、生成器は毎秒約64/2.62 ≈ 24.4個の新しい接続を開きます。接続ごとにスレッドを使うと、毎秒約 24 回のスレッドの開始と 24 回の終了になります。このレートでの 10 秒の実行では、1,000,000 件のリクエストを測定します。その p99.9 を超えるのはそのうち 1,000 件、毎秒 100 件で、p99 を超えるのは毎秒 1,000 件です。入れ替わりのたびに、他のスレッドで処理中の約100/24.4 ≈ 4件のリクエストが 1 ミリ秒未満だけ遅れるとすると、遅れたリクエストだけで上位 0.1% が埋まり、上位 1% は依然として通常の待ち行列でできています。まれで周期的な停滞は、p99 を動かすよりずっと前に遠い裾を動かします。このような欠陥を見るのに p99.9 とすべてのリクエストのタイマーが必要なのはこのためです。

Sibuna と Anubis

この章の学習目標 同じ負荷の下で、Sibuna の数値を確立した製品の数値と並べて読み、どの違いが製品から、どの違いが構成から来ているかを述べます。

Techaro の Anubis は、Sibuna の Gate と同じように Web サイトの前に置く、広くデプロイされている作業証明のゲートです。Anubis v1.27.0 の公式リリースバイナリ（`anubis-1.27.0-linux-amd64`、40,001,698 バイト）を、同じホスト、同じ CPU、同じワークロードで Sibuna と並べて測定しました。二つの結果ファイルは次のコマンドで記録しました。

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

Anubis は、このプロジェクトの以前の Python のドライバーと同じように起動しました。`--bind`、`--metrics-bind`、`--target=`（forward-auth モードでは空、リバースプロキシモードではオリジン）、`--difficulty 2`、`--slog-level ERROR`、`--cookie-secure=false`、そして `Mozilla` を含むすべての User-Agent にチャレンジし、DNSBL とハニーポットを無効にしたポリシーです。`GOMAXPROCS` は Sibuna のワーカー数と等しく、どちらの製品のスレッドもすべて同じ CPU セットに固定します。Hashcash の作業量はそろえています。Sibuna は 8 個のゼロビット、Anubis は 16 進数で 2 個のゼロで、期待ハッシュ回数はどちらも 256 回です。ハーネスは各製品のセッションをブラウザーと同じように得ます。Anubis では、チャレンジのページを取得し、`<script id="anubis_challenge">`からチャレンジを読み、`SHA-256(randomData || nonce)` が `00` で始まる nonce を見つけ、ページの検証用 Cookie とともに `pass-challenge` に渡します。forward-auth モードでは、`admitted` と `challenged` のワークロードは Anubis の確認エンドポイント（`/.within.website/x/cmd/anubis/api/check`）に問い合わせます。

製品全体のワークロード

四つのワークロード、負荷、統計は製品全体のハーネスと同じですが、一つ変更があります。オリジンは、以前のドライバーと同じく、負荷用の CPU に固定した `caddy respond` です。Anubis のリバースプロキシの実行では何万ものソケットが残るので、ハーネスは毎回の実行の前に、`TIME_WAIT` のソケットが 3,000 未満になるまで待ちます。

表 18 forward-auth モード：5 秒の `wrk` の実行 3 回の中央値（`benchmarks/results/comparison-latest.md`、2026-10-16T00:55:19Z 生成）。

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

表 19 リバースプロキシモード、`caddy respond` の背後：5 秒の `wrk` の実行 3 回の中央値（`benchmarks/results/comparison-latest.md`、2026-10-16T00:55:19Z 生成）。

ワークロード	数値	Sibuna Gate	Sibuna Shield	Anubis 1.27.0
admitted	req/s（ステータス）	106,376 (200)	103,298 (200)	19,398 (200)
	p99 us	3,331	3,321	11,062
	CPU us/req	27.02	27.92	190.40
	ピーク RSS KiB	16,528	16,444	39,980
challenged	req/s（ステータス）	198,402 (200)	197,327 (200)	28,842 (200)
	p99 us	325	328	9,546
	CPU us/req	13.05	13.78	137.73
	ピーク RSS KiB	16,544	16,464	516,872
allowed_static	req/s（ステータス）	105,427 (200)	106,819 (200)	28,862 (200)
	p99 us	3,320	3,316	30,124
	CPU us/req	26.24	26.52	135.41
	ピーク RSS KiB	16,544	16,464	616,548
attack	req/s（ステータス）	105,616 (200)	210,483 (403)	23,399 (200)
	p99 us	3,327	306	14,124
	CPU us/req	26.98	12.28	169.04
	ピーク RSS KiB	16,544	16,464	599,516

forward-auth モードでは、Sibuna は毎秒 193,087 から 212,569 リクエストを p99 304 から 342 us で処理し、Anubis は 36,509（admitted）から 113,170（challenged）を p99 3,781 から 7,477 us で処理しました。Sibuna の稼働中のコア数は四つのうち 2.49 から 2.90 にとどまったので、上限を決めたのは負荷生成器です。Anubis は 3.73 から 3.88 で、CPU 律速でした。最も明確な数値はリクエストあたりの CPU 時間です。許可されたリクエストのコストは、Sibuna Gate で 12.09 us、Anubis で 106.30 us でした。この差の一部はおそらくセッショントークンによるものです。Anubis は既定で Ed25519 でセッションに署名し、Sibuna は鍵付き BLAKE3 の MAC（プリミティブの表で 155.8 ns）を確認します。後の許可の測定では、HS512 のトークンにすると Anubis のセッションの確認が半分以上増えており、この読みを支持しています。リバースプロキシモードでは、Sibuna のプロキシする行は、稼働中のコア数 2.77 から 2.98 で毎秒 103,298 から 106,819 リクエストを処理しました。これは product-latest.json のスタブのオリジンでの 138,965 から 147,303 を下回り、デーモンは CPU 律速ではないので、上限はデーモンの外にあります。オリジンの caddy respond が負荷用の CPU を wrk と共有しているのです。その裾は 2.1 から 2.4 ms の p90 に現れています。Anubis は稼働中のコア数 3.68 から 3.97、リ

クエストあたりの CPU 135 から 190 us で、毎秒 19,398 から 28,862 リクエストを処理しました。その常駐メモリは、admitted のワークロードの 39,980 KiB から、challenged で 516,872 KiB、allowed_static で 616,548 KiB に増え、ワークロードの後もアイドル時の 21,552 KiB に対して 589,276 KiB を保持していました。増加は challenged で始まりました。そこでは各リクエストが独自の ID を持つ新しいチャレンジを受け取ります。これは発行したチャレンジごとにレコードを保持していることを示唆しますが、ハーネスはその原因を確認していません。Sibuna のピークは、両方のモードのどの行でも 16,228 から 16,544 KiB にとどまりました。
条件がそろっていない行もいくつかあります。

- forward-auth モードでは、Anubis は/robots.txt と attack のパスに、オリジンなしで自ら 200 で応答します。Sibuna は、そのモードでの他のリクエストと同じように応答します。
- Anubis にはリクエストの検査がないので、attack のワークロードの SQL インジェクションを許可します。Shield は、より多くの作業の後で 403 で拒否します。Gate は Anubis と同じく許可します。
- 製品はセッションの署名方法が異なり（Anubis は既定で Ed25519、Sibuna は鍵付き BLAKE3 の MAC）、描画するチャレンジのページも異なります。

許可処理の操作

許可のハーネスは、許可を構成する四つの操作を、forward-auth モードで、各製品に二つの CPU を与えて計時します（Sibuna のワーカーは二つ、GOMAXPROCS=2）。負荷用の CPU 上の二つのクライアントスレッドが、それぞれ一つの keep-alive 接続を持ち、各バッチを分担します。バッチは 200 操作、検証については 64 個の証明で、バッチの前に用意します。7 バッチの中央値を報告します。

- セッションの確認：有効なセッションでの確認エンドポイント（200）。
- 未認証の確認：Cookie なしの同じ確認（401）。
- チャレンジの開始 :ブラウザーがチャレンジを始めるために取得するもの。Sibuna では中間ページとチャレンジレコードの二つのリクエスト、Anubis ではチャレンジを埋め込んだページの一つです。
- 証明の検証：操作ごとに一つの新しい証明。Sibuna の証明のチャレンジは 25 ms 間隔で発行します。これは適応的な難易度がビットを加え始める毎秒 50 件を下回るので、そろえた作業量が保たれます。Sibuna は--challenge-rate-limit 100000000 で実行するので、リミッターは確認されますが、使い果たされることはありません。

Anubis は、既定の Ed25519 のトークンと、--hs512-secret を指定した場合の 2 回実行しました。

表 20 毎秒の許可操作数、二つのクライアント、製品は二つの CPU（benchmarks/results/admission-latest.md、2026-10-16T00:55:44Z 生成）

製品（トークン）	セッションの確認	未認証の確認	チャレンジの開始	証明の検証	実行後の RSS、KiB
Sibuna（鍵付き 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

製品（トークン）	セッションの確認	未認証の確認	チャレンジの開始	証明の検証	実行後の RSS、KiB
Anubis 1.27.0（HS512）	16,976	27,785	7,952	19,261	29,840

7 バッチの範囲はファイルにあります。Sibuna の範囲は中央値から 11% 以内に収まります。Anubis の範囲はより広く（Ed25519 のトークンでの未認証の確認は 25,394 から 32,741）、200 操作のバッチは 2 から 25 ms しか続かないので、これらの数値は 1 割程度の精度で読みます。遅いほうのクライアントの操作あたりの時間が往復時間を表します。Sibuna のセッションの確認で 22.7 us、Anubis の Ed25519 のトークンでの確認で 182.5 us、HS512 で 117.7 us です。上のどの行でも、Anubis が Sibuna より多くのリクエストを処理したり、少ない CPU やメモリしか使わなかったりしたことはありません。これはこのホストでのこれらのワークロードについての記述であり、それ以上のものではありません。

Anubis の行の読み方　Anubis は有能で広くデプロイされている製品であり、これらの行は Anubis が遅いという主張ではありません。Anubis はガベージコレクション付きのランタイムで動き、検証に共有の秘密が要らないように既定で Ed25519 でセッションに署名し、複数のチャレンジ方式と豊かなポリシー言語を提供し、チャレンジの状態をストアに保持します。これらは、このハーネスが測らない利点を持つ設計上の選択です。行が示すのは、2026 年 10 月に、一つのホストで一つの負荷の形の下で、リクエストが各製品にかけたコストです。Sibuna は新しく、これは確立した製品と並べた最初の測定です。

コンソールのオーバーヘッド

SID 0007 は、コンソールの分離に関する取り決めを定めています。コンソールを有効にしたときのコストは、データプレーンのスループットの最大 1%、p99 のレイテンシーの最大 10% までです。測定は、構成のマトリクス（組み込まない、組み込むが無効、アイドル、ダッシュボード八つが稼働中）とワークロードについて、交互に繰り返す実行で行い、不確かさを報告します。決定的でない実行やノイズの多い実行では適合を示せず、プリミティブの基準値や簡易な確認の実行でこの判定の代わりにすることはできません。

SID 0010 は一つの測定を記録しています。コンソールを組み込んだ場合、無効にした場合、ストレージのみの場合、有効でアイドルの場合のいずれでも、16 個の製品のワークロードはコンソールなしでビルドしたデーモンの約 1% 以内に収まり、これはノイズの範囲内です。この記録は結果ファイルを示していません。負荷の下でダッシュボードを八つ稼働させた構成は測定していないので、SID 0007 の正式な判定はまだ完了していません。

八つのダッシュボードの実行が確かめるのは、オーバーヘッドではなく機能です。`sibuna-browser dashboards` は、一つの SQLite ノード上で、コンソール、四つのワーカー、そして `/static/*`、`/about`、`/missing/*`、`/broken/*` を許可し、それ以外にはすべてチャレンジするポリシーでリリースビルドのデーモンを起動しました。オリジンのフィクスチャは `/missing/*` に 404、`/broken/*` に 500 で応答します。生成器は 30 秒間、毎秒 200 リクエストを送り、10 個のパス（一つは SQL インジェクション）、三つの User-Agent、500 個の転送アドレスを順に使いました。各ダッシュボードは専用の Chromium で専用のセッションを持って動きました。ダッシュボード 1 から

3 は管理者のセッション、4 から 8 は最初のダッシュボードが発行したコードで開いたウォールディスプレイです。コンソールは、アドレスごと、アカウントごとに毎分 5 回のサインインしか許さないからです。

表 21 一つのノードでの八つの Traffic ダッシュボード（benchmarks/results/console-8-dashboards-2026-10-15.md、2026-10-15 記録、Chromium 155.0.8059.39）。結果：PASS

項目	値
データプレーンのリクエスト	30.0 秒で 6,000 件（毎秒 200 件、4スレッド、リクエストごとに一つの接続）
応答あり	6,000: 200 x 1,200, 401 x 3,000, 403 x 600, 404 x 600, 500 x 600
エラー（応答なし）	0
レイテンシー p50 / p90 / p99 / max	246 / 311 / 373 / 552 us（接続を含む）
開始時のデーモンの VmRSS	36,152 KiB
ダッシュボードを 8 つ開いたときのデーモンの VmRSS	38,028 KiB
トラフィック後のデーモンの VmRSS	41,496 KiB（VmHWM 41,496 KiB）

タイルは、トラフィックの間は 2 秒ごとに、その後は 3 秒後に読みました。どのダッシュボードも同じ合計（Requests 6,000、Admitted 2,400、Challenged 3,000、Policy denied 600、Origin 4xx 600、Origin 5xx 600）で終わり、134 から 139 個の stats の差分を受け取り、WebSocket を開いたままにし、例外、コンソールのエラー、Content-Security-Policy の違反を一つも報告しませんでした。ブラウザーの購読者八つは、コンソールの上限 64 を大きく下回ります。この実行のデーモンは、製品のハーネスのメモリ上のデーモンと違って SQLite ストレージとコンソールを持つので、その常駐メモリは製品のハーネスのものとは比較できません。

測定していないもの

表 22 この部が測定しないものと、測定の代わりになるもの

対象	ここで測定しない理由	代わりになるもの
SafeLine コミュニティ版 9.x	Docker Compose のスタックとしてしか配布されていません。ホストには Docker デーモンがなく、検出器のイメージはそこでコンパイルできるオープンソースのビルドではありません	公表された、ベンダーが報告する事実（2026 年 9 月に確認）：検出率 71.65%（balance）と 76.17%（strict）、誤検知 0.07%（balance）と 0.22%（strict）。最小のホストは SSSE3 付きの Linux x86_64 または arm64、CPU 1 コア、RAM 1

対象	ここで測定しない理由	代わりになるもの
		GB、ディスク 5 GB、Docker 20.10.14 以降、Compose 2.0 以降
Cloudflare WAF	ホスト型のサービスです。ループバックでの測定は WAF ではなくネットワークの経路を測ることになります	公表されたプランごとの上限（2026 年 9 月に確認）：カスタムルールは Free 5、Pro 20、Business 100、Enterprise 1000。レート制限ルールは Free 1、Pro 2、Business 5、Enterprise 100
負荷の下のクラスター	一つの PostgreSQL データベースで複数のノードを動かすハーネスはまだありません	SID 0011 が規定するプロトコルの性質（一つの共有シード、発行者に結び付いたチャレンジ）と、三つのノードによるその機能テスト（criterion 2）。測定はなし
リクエストの経路上のストレージ	どのハーネスでも、両方のデーモンはメモリ上で動きます	SID 0011：一つのストレージスレッドがすべての SQL を所有し、リクエストの経路がそれを待つことはありません
TLS、WAN、本物のブラウザー	ハーネスはループバックと HTTP を使います。セッションは各ケースで一度だけ `sibuna-pow` で解きます	なし。デプロイごとに自分の経路を測ります
ブラウザーのソルバーの解答時間	ハーネスはネイティブのソルバーで 8 ビットで解きます	上のモジュールのサイズ。ネイティブの所要時間がブラウザーでの解答時間の代わりにならない理由は第 VII 部で説明します
割り当ての回数	ハーネスはアロケーターを計装しません	第 V 部の定常状態のテスト
負荷の下でダッシュボードを八つ稼働させたときのコンソールの分離	SID 0007 の正式な判定はまだ実施していません	SID 0010 の数値（コンソールがアイドルで約 1% 以内）と、上の機能の実行

この表の行は、製品の性能についての主張ではありません。SafeLine 自身の数値はスループットではなく検出の質に関するもので、Cloudflare はリクエストあたりのコストではなくプランごとの上限を公表しています。Docker ホストがあれば、誰でも同じ `wrk` のワークロードの背後で SafeLine を実行できます。

演習 8.1. 別のマシンで `cargo xtask bench primitives` を実行し、Hashcash の検証とトークンの検証の行を基準のホストと比べてください。二つのうち、ハードウェアのハッシュ命令に最も依存するのはどちらでしょうか。それはなぜでしょうか。

演習 8.2. forward-auth の表で、ある行の稼働中のコア数を計算してください。リクエストあたりの CPU マイクロ秒と毎秒のリクエスト数の積を 100 万で割った値です。デーモンは CPU 律速だったでしょうか。次に、そのリクエストあたりの CPU マイクロ秒を最初の表の分類コストと比べてください。リクエストあたりの CPU コストがはるかに高いのはなぜでしょうか。その差はどの構成要素によるものでしょうか。
ヒント：`accept` から、フラッシュした応答までの間で、プリミティブのスイートが計時しないものをすべて挙げてください。

不変条件を説明してください。 この部のどの数値も、繰り返しにわたる中央値で、そのばらつきを並べています。バッチの中央値による数値とリクエストごとの p99.9 の違い、プリミティブの表が前者を、レイテンシーの表が後者を報告する理由、そして上で述べたスレッドの入れ替わりによる停滞を明らかにできたのが後者だけである理由を説明してください。

第 I X 部

運用とデプロイ

二つの運用構成、すべてのコマンドラインフラグ、*forward-auth* のレシピ、ポリシーファイル、*SQLite* または *PostgreSQL* による永続ストレージ、一つの共有データベースによるクラスタリング、パッケージングを扱います。

運用構成と設定

この章の学習目標　この章を読み終えると、Sibuna をリバースプロキシまたは forward-auth の検証器として実行し、運用構成を選び、作業証明の層と難易度を設定し、マスターシークレットを管理できるようになります。

運用構成の選び方

- *Gate*（`--gate`、別名`--no-waf`）：作業証明による許可、セッション、宣言的なルール、レピュテーション、GCRA による制限、禁止。
- *Shield*（既定、`--shield`、別名`--waf`）：Gate に意味解析ファイアウォールを加えたもの。
- 分散デプロイ（どちらの運用構成でも選択可能）：`--data-dir` は、ポリシー、レピュテーション、フォレンジックをローカルの SQLite ファイルに保持します。`--database-url` と `--cluster-node` は、複数のノードを一つの PostgreSQL データベースに接続します（SID 0011）。

コマンドラインのリファレンス

この表は `sibuna --help` のすべてのオプションを示します。範囲はパーサーが強制するものです。

表 23 デーモンのオプション

フラグ	既定値	意味
`--port, -p`	`8080`	待ち受けポート、1 から 65535
`--host, -h`	`0.0.0.0`	待ち受けアドレス、IP アドレスのリテラル
`--upstream-host`	`127.0.0.1`	オリジンのホスト（リバースプロキシモード）
`--upstream-port, -u`	`3000`	オリジンのポート
`--mode, -m`	`reverse_proxy`	`reverse_proxy` または `forward_auth`（`proxy` と `auth` も可）、一度だけ指定
`--workers, -w`	CPU ごとに一つ	待ち受けソケットを共有する受付スレッド、0（CPU ごとに一つ）から 64。受け付けた各接

フラグ	既定値	意味
		続にはその後、専用のスレッドが割り当てられます
`--max-connections`	`1024`	同時に処理する接続、1 から 8192。それを超える接続には `503` で応答します
`--max-connections-per-ip`	`64`	一つのソケットの相手アドレスが保持できる接続、0（上限なし）から 8192。IPv6 の相手は`--ipv6-prefix` のネットワーク単位で数え、転送ヘッダーは関与しません。それを超える接続には `503` で応答します
`--idle-timeout`	`15`	最長の無通信時間（秒）、0 から 86400。リクエストのヘッダー部はこの時間内に届かなければなりません。オリジンの応答は読み込みのたびにこれを更新するので、遅いストリームもバイトが流れている間は生き続け、無通信のオリジンは両方のソケットで切断されます。プロキシするアップロードは、期間ごとに 16 KiB を届けなければなりません（遅いボディーの攻撃に対する最低速度）。クライアントのソケットには同じ長さの書き込みタイムアウトがあるので、その間 1 バイトも受け取らないクライアントは接続を失います
`--websocket-idle-timeout`	`300`	アップグレードした接続のアイドル期限（秒）、0 から 86400
`--trust-forwarded`	無効。forward-auth では有効	相手からの `X-Forwarded-For`/`X-Real-IP` を受け入れる
`--no-trust-forwarded`	無効	forward-auth モードでも、決して受け入れない
`--trusted-hops`	`1`	転送された識別情報を信頼する場合、クライアントは右からこの位置にある `X-Forwarded-For` の項目、1 から 8
`--ja4-header`	無効	信頼するイングレスがクライアントの JA4 TLS フィンガープリントを送るリクエストヘッダー（たとえば `cf-ja4`）。トークン文字で 1 から 64 文字、`X-Forwarded-*`、`X-Real-IP`、`X-Sibuna-*`、`Forwarded` 以外。転送された識別情報

フラグ	既定値	意味
		を信頼する場合にだけ読む（「イングレスから受け取る JA4 フィンガープリント」を参照）
`--ipv6-prefix`	`64`	レート制限、禁止、クライアントごとの接続数の上限を適用する IPv6 の接頭辞長、32 から 128
`--algorithm, -a`	`posw`	`posw` または `hashcash`
`--difficulty, -d`	`16`	作業ビット、1 から 64。Hashcash のゼロビット数、または PoSW の深さに 3 を足した値
`--posw-challenges`	`16`	逐次作業の証明あたりの開示数、1 から 32
`--token-scheme`	`mac`	`mac`（鍵付き BLAKE3）または `ed25519`
`--token-ttl`	`86400`	セッションの有効期間、秒（1 から 31,536,000）
`--challenge-ttl`	`300`	チャレンジの有効期間、秒（1 から 86,400）
`--secret-file, -s`	ランダム	16 進数 64 文字または生の 32 バイト。`SIBUNA_SECRET`（16 進数 64 文字）も受け付けます
`--cookie-name`	`__sibuna_token`	セッション Cookie の名前、トークン文字で 1 から 64 文字
`--secure-cookie`	無効	`Secure` 属性を付加
`--gate` / `--shield`	shield	運用構成の選択
`--rate-limit`	`100`	クライアントごとの窓あたりのリクエスト数（GCRA のバースト）
`--rate-window`	`10`	窓の秒数、1 から 86400
`--challenge-rate-limit`	`30`	クライアントごとの窓あたりのチャレンジの発行と検証の回数。リクエストの予算とは別
`--ban-seconds`	`3600`	ハニーポットによる禁止の期間、0 から 31,536,000
`--policy-file, -P`	なし	宣言的な JSON のポリシー
`--rate-table-cells`	`262144`	レートリミッターの表あたりのセル数、各 16 バイト、4,096 から 16,777,216、2 の累乗に切り上げ
`--ban-table-slots`	`65536`	禁止テーブルのスロット数、各 24 バイト、1,024 から 16,777,216、2 の累乗に切り上げ

フラグ	既定値	意味
--spent-tags	1048576	リプレイ対策として記憶する解かれたチャレンジ、各 24 バイト、65,536 から 16,777,216
--data-dir, -D	なし	一つのノード：SQLite ファイル<path>/sibuna.db に永続化
--database-url	なし	クラスター：共有の PostgreSQL データベース（postgres://または postgresql://）
--database-url-file	なし	所有者だけが読めるファイルからデータベースの URL を読む
--cluster-node	なし	このノードの ID。チャレンジ鍵とインシデントの識別子の上位ビットに結び付きます。データベースの URL とともに必須
--storage-poll-ms	500	ストレージのティック間隔、1 から 60,000 ms
--verbose, -v	無効	詳細なログ出力
--help / --version		ヘルプ（コンソールを組み込んでいる場合はコンソールのオプションが先）またはバージョンを表示して終了

--data-dir とデータベースの URL は互いに排他的で、--database-url と --database-url-file も互いに排他的です。--cluster-node のないデータベースの URL は拒否されます。Zaxonlite のクラスターオプション --cluster-listen、--cluster-peer、--cluster-secret-file、--cluster-tls-cert、--cluster-tls-key、--cluster-tls-ca は廃止されました。どれかを指定すると、起動は RemovedOption と SID 0011 を示すヒントで停止します。コンソールのオプションは次の章に挙げます。

> **マスターシークレット** --secret-file も SIBUNA_SECRET もなければ、デーモンはランダムなシードを引き、バナーに"random (set –secret-file)"と表示します。その場合、トークンとチャレンジはプロセスとともに失われ、ノードが異なるシードを持つクラスターは互いの Cookie を拒否します。head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret で生成し、モードを 0600 にしてください。

コンソールのプレビュー

コンソールは既定の console フィーチャーでデーモンに組み込まれますが、起動するのは--console を指定したときだけです。コンソールにはストレージが必要です。--data-dir も--database-url もない--console は CONSOLE001（StorageRequired）で拒否されます。コンソールを除いてコンパイルするには cargo build --release -p sibuna --no-default-features --features sqlite を使います。--no-default-features だけのビルドには、ストレージもコンソールもありません。デーモンを停止した状態で、ローカルで初期設定します。

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

sibuna console init-admin も同じコマンドです。http://127.0.0.1:19446/console/を開き、生成された一時パスワードを変更します。一時パスワードは最初のサインインで変更しなければならず、1 時間で失効します。サインイン後のナビゲーションでは、アニメーションする国の地球儀、リクエストとチャレンジの統計、インシデントの調査、ポリシーと検査のエディター、ユーザー、スコープ付き API トークン、監査、配信ノードの制御を利用できます。認証画面は地球儀の形状もテレメトリーも読み込みません。インターフェースは、デーモンのビルドスクリプトが cmd/console-ui（クレート sibuna-console-ui）からビルドする WebAssembly モジュールです。これとコミット済みのスタイルシートは、通常のビルドに含まれます。

表 24 コンソールのオプション

フラグ	意味
--console <ip:port>	このリスナーでコンソールを起動（IPv6 は角括弧で囲み、ポートは 0 以外）。--data-dir または--database-url が必要
--console-key-file <path>	16 進数 64 文字、所有者のみの権限。保存した二要素認証と Webhook の秘密を保護
--console-origin <https-origin>	プロキシの背後での正規の HTTPS オリジン
--console-behind-proxy	TLS を終端するプロキシがすべてのリクエストを届ける
--console-trusted-proxy <cidr>	接続が転送フィールドを運んでよいネットワーク。繰り返し指定可、最大 16
--console-advertise <origin>	他のコンソールがこのコンソールのために表示するリンク
--console-location <latitude,longitude>	このノードの地球儀上の宣言位置
--console-probe <node-id>=<http://ip:port>	ヘルスチェックするピアのデータプレーンのリスナー。繰り返し指定可、最大 8
--console-peer <node-id>=<https://origin>	管理用のピア。繰り返し指定可、最大 8
--console-peer-key-file <path>	ピア鍵：所有者のみ、16 進数 64 文字、コンソール鍵とは独立
--console-peer-ca-file <path>	ピア用の任意の PEM のトラストアンカー。システムのルートを置き換える
--console-query-steps <n>	集計の読み込みあたりの SQLite のステップ数、100,000 から 50,000,000、既定 4,000,000

フラグ	意味
`--console-capture-heads`	インシデントごとに、伏せ字にしたリクエストとオリジンの応答のヘッダー部を保存。既定では無効
`--console-capture-header <name>`	保存するヘッダー部にこのヘッダーの値を残す。繰り返し指定可、最大 16。`--console-capture-heads` が必要
`--console-cookie-secure`	プロキシなしでコンソールの Cookie に `Secure` 属性を付ける（パーサーは受け付けるが、`--help` には載らない）

`--console` なしのコンソールのオプション、繰り返したオプション、不正な形式の値は、`CONSOLE002` とオプションのエラー名で起動を停止します。コンソールの規則に反する設定（たとえば信頼するプロキシのない、ループバック以外のリスナー）は、`CONSOLE001` で起動を停止します。

コンソールの各画面

以下のスクリーンショットは、SQLite ストレージを持つループバック上の一つのノードで動くリリースビルドから撮ったものです。ポリシーは既定のチャレンジの上に四つの `ALLOW` ルールを置いたもので、約 4 分間の合成トラフィックの後に撮りました。トラフィックは、文書用のアドレス範囲からのアドレス、SQL インジェクションとクロスサイトスクリプティングの試行、三回のハニーポットへのアクセス、ヘッドレス Chromium で解いた七つのチャレンジです。したがって、どの数値も説明のためのものです。撮影には `cargo xtask browser console --viewport-only` を使いました。これは Chromium で 1440×960 ピクセルの画面でコンソールを巡回し、すべてのページを判定します。各ページはタイトルに書いた一つの問いに答え、製品、配信ノード、ページ、戻る手段を同じ場所に置いています。

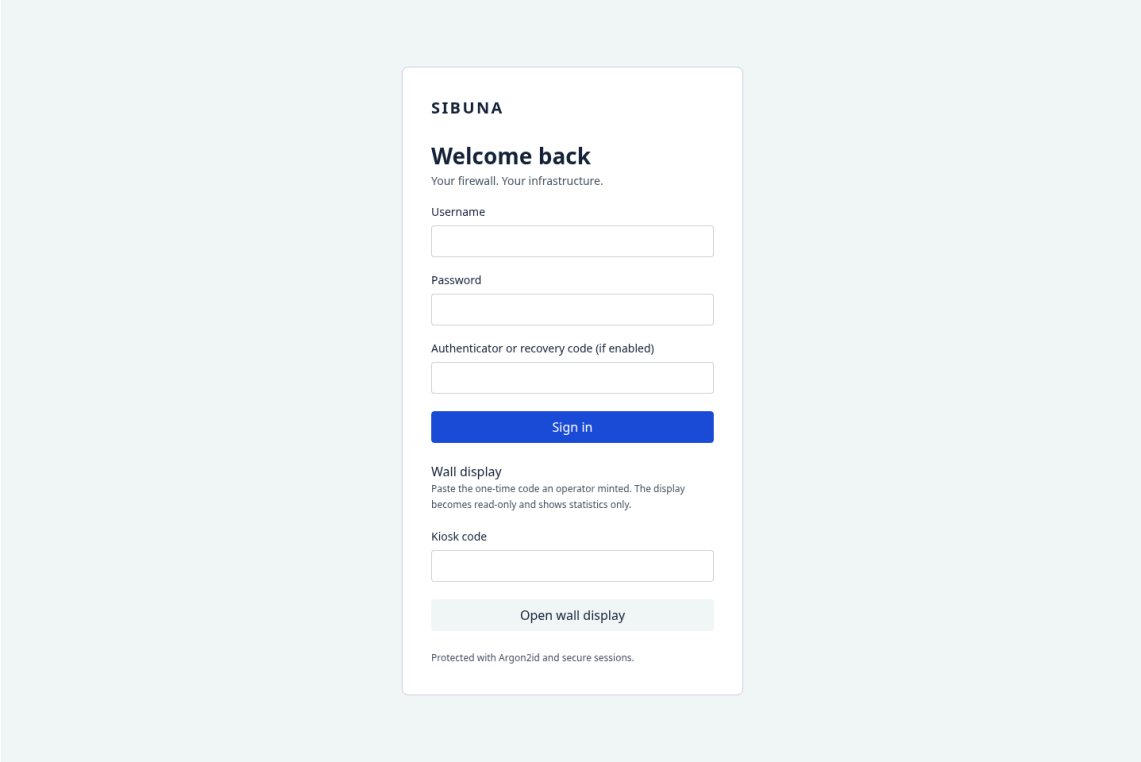


図 13 サインイン。公開された画面は、地球儀の形状、GeoIP のデータ、テレメトリーのいずれも読み込みません。二つ目のフォームは、ウォールディスプレイ用のコードを読み取り専用の統計セッションと交換します。

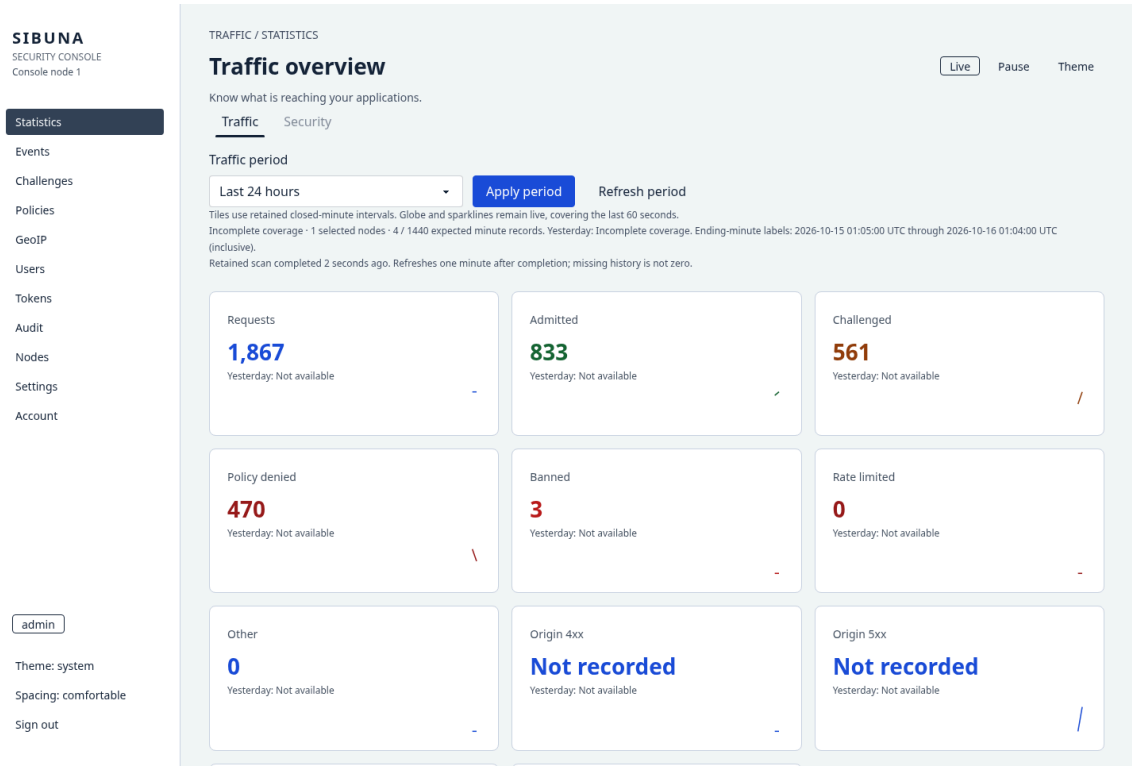


図 14 統計（Statistics）の Traffic、直近 24 時間。期間の選択の下の注記が観測範囲を示します。期待される 1,440 個の分単位のレコードのうち四つしかないので、"Yesterday"はすべて"Not available"と表示されます。タイルは、保持している確定済みの分のカウントを、ライブの 60 秒のスパークラインとともに示します。判断の色（許可は緑、チャレンジは琥珀色、拒否は赤）はどのパネルでも同じです。保持している期間のオリジンの 4xx と 5xx は"Not recorded"と表示されます。分単位のレコードは、オリジンのカウントに意味を与えるプロキシのモードを保持していないからです。ライブの起動以降の合計にはそれらが表示されます。見えているタイルの下には、二つの水準のタイル（有効な禁止の項目と正常なノード）、タイムライン、そしてリクエストのパスと参照元のホストを順位付けし、クライアントの OS、ブラウザー、応答のステータスを数えるサンプリングしたパネルがあります。

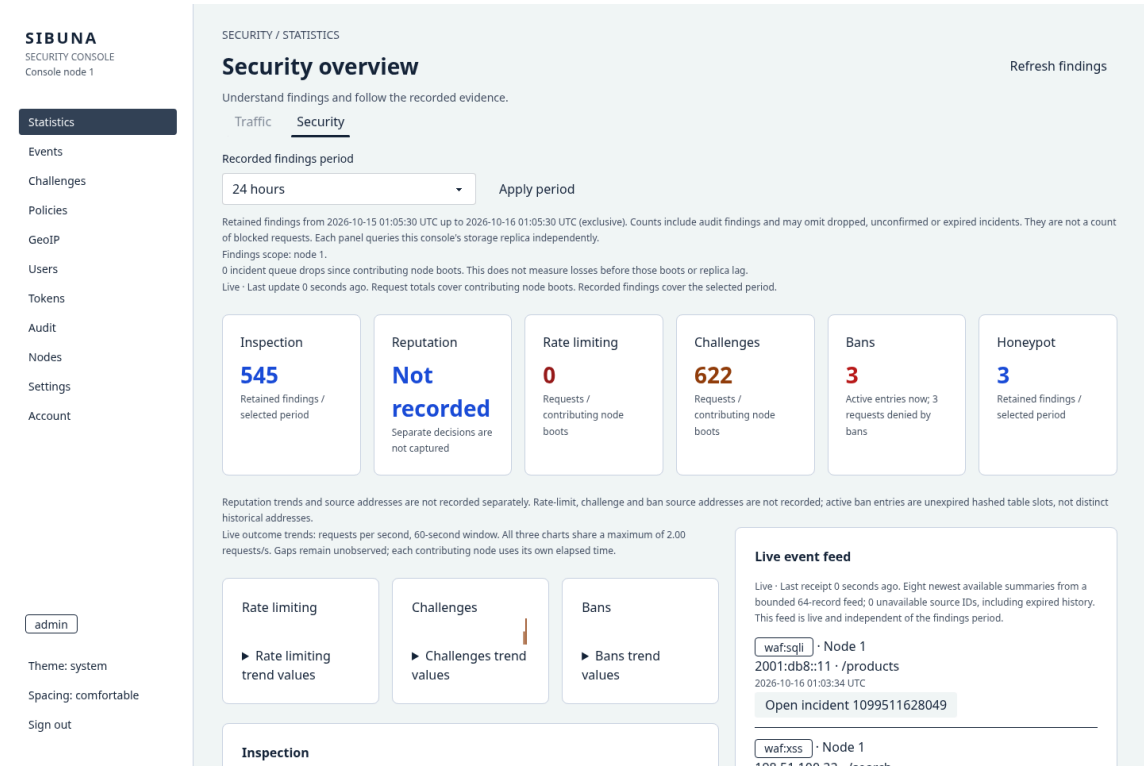


図 15 統計（Statistics）の Security。選択した期間についてのモジュールごとのタイル、ライブの結果の傾向、最新の検出（ここでは `waf:sqli` と `waf:xss`）を示すライブのイベントフィードです。記録されていない値は、ゼロを表示する代わりにそのことを示します。Reputation のタイルは"Not recorded"と表示されています。ページのさらに下には、60 個の等しい区間ごとの検出、攻撃のカテゴリー、攻撃されたパスがあります。

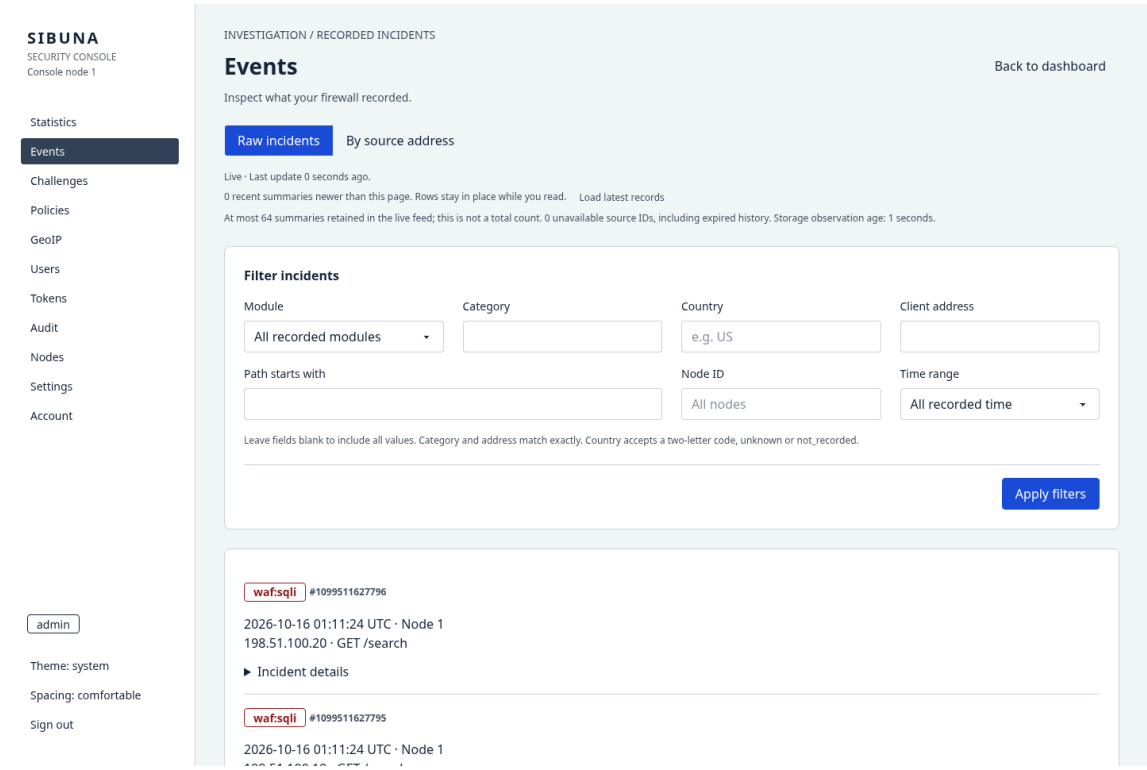


図 16 イベント（Events）。記録されたインシデントの上にフィルターのフォームがあり、インシデントは新しい順に並びます。各インシデントはその場で開き、証拠を示します。選ばれたローカルの応答、バイト長、キャンペーンの候補、そして一致したルール、スコアの項、JA4 フィンガープリントについての明示的な"not recorded"の項目です。`--console-capture-heads` を指定すると、伏せ字にしたリクエストのヘッダー部（リバースプロキシを通して監査付きで許可されたものでは、オリジンの応答のヘッダー部も）を要求に応じて開けます。UTF-8 または Latin-1 として表示し、応答の状態（取得済み、ローカル、forward-auth で未観測、オリジン利用不可）を示し、cURL コマンドとしてコピーできます。保存するのは列挙したヘッダーの値だけで、`--console-capture-header <name>`でその一覧に加えます。それ以外の場合、ヘッダー部は記録なしと表示されます。拒否と許可の操作は、アドレスを入力済みにした IP グループのフォームを開きます。

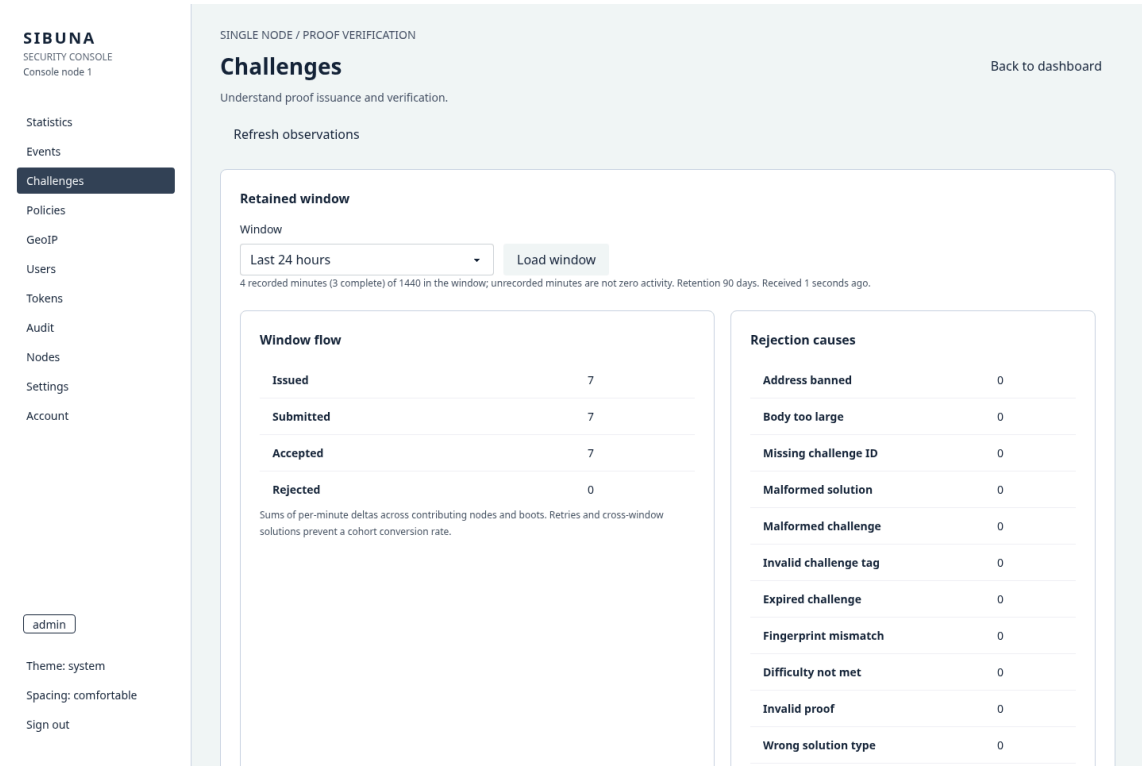


図 17 チャレンジ（Challenges）。保持している窓（1 時間、1 日、7 日）は永続的な分単位のレコードを合計し、何分が記録され完全かを示します。ここでは七つのチャレンジが発行、提出、受理され、拒否の原因はすべてゼロです。さらに下には、今回の起動以降のライブの合計、設定したパラメーターと最近発行したパラメーター、アルゴリズムとパラメーターの区分で分けた受理された解答時間のヒストグラム、適応的な難易度の遷移、拒否と許可のショートカット付きのアドレスごとのレコード（7 日間保持）があります。

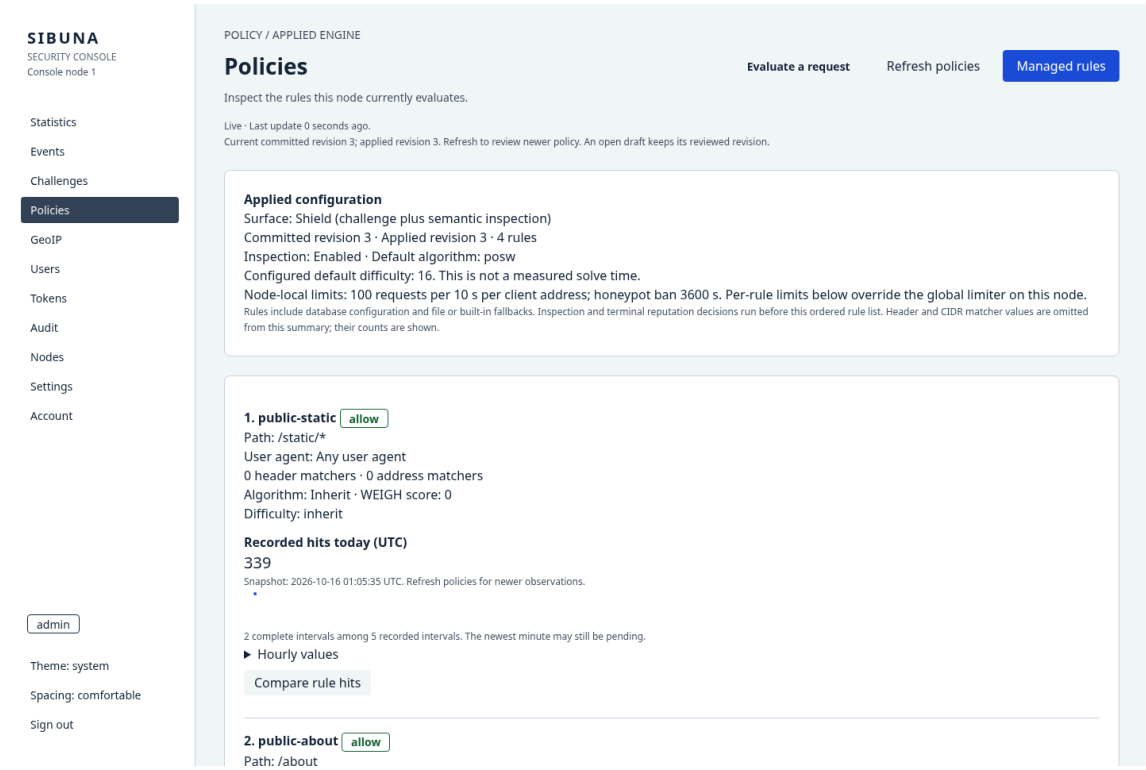


図 18 ポリシー（Policies）。適用中のエンジンが、その運用構成、リビジョン、既定のアルゴリズムと難易度、ノードにローカルな制限を示し、続いてルールを評価順に、今日のヒット数とそのカウントの観測範囲とともに挙げます。さらに下には、検査モードのマトリクス、リクエストのテスター、IP グループがあります。*Managed rules* でエディターが開きます。編集を保存すると、まず変更したフィールドのレビューを示し、確定すると候補全体を検証して新しいリビジョンを作ります。

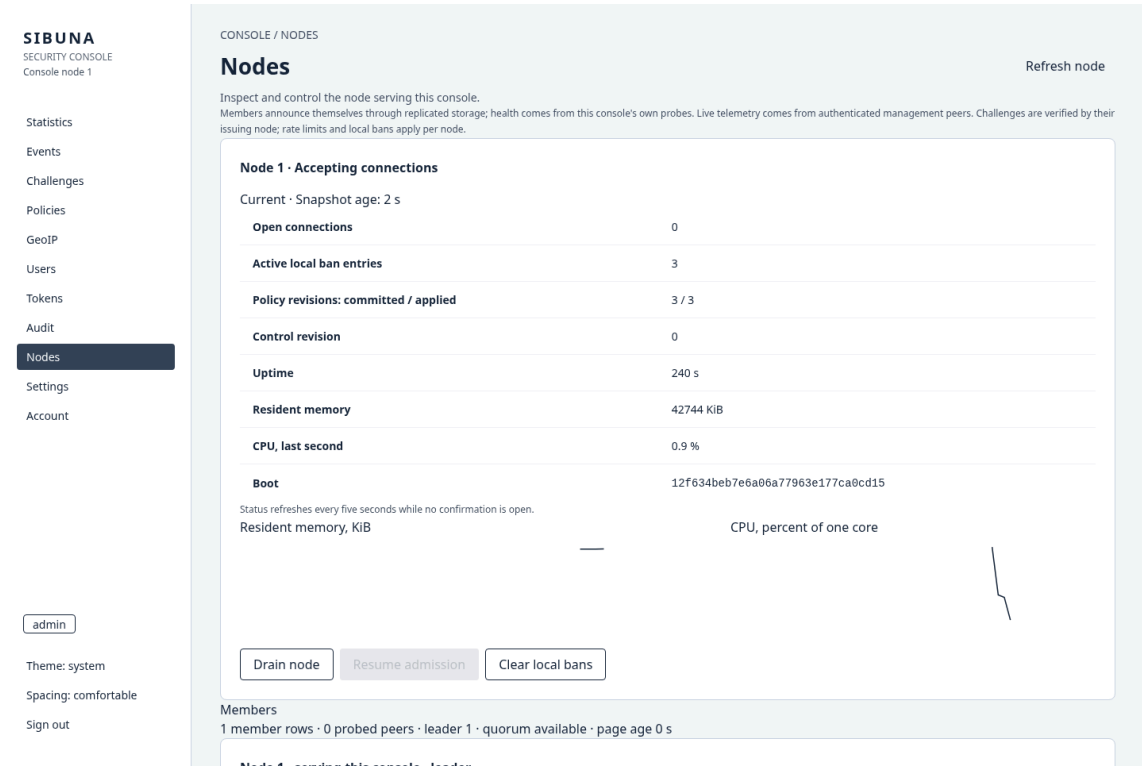


図 19 ノード（Nodes）。配信ノードの状態とコマンド、直前 1 秒間の常駐メモリと CPU、その 60 秒のスパークラインです。その下に、通知されたすべてのメンバーを、コミット済みのリビジョンに対する適用中のポリシーのリビジョン、管理用のアドレス、プローブの結果とともに示します（SID 0007、SID 0013）。

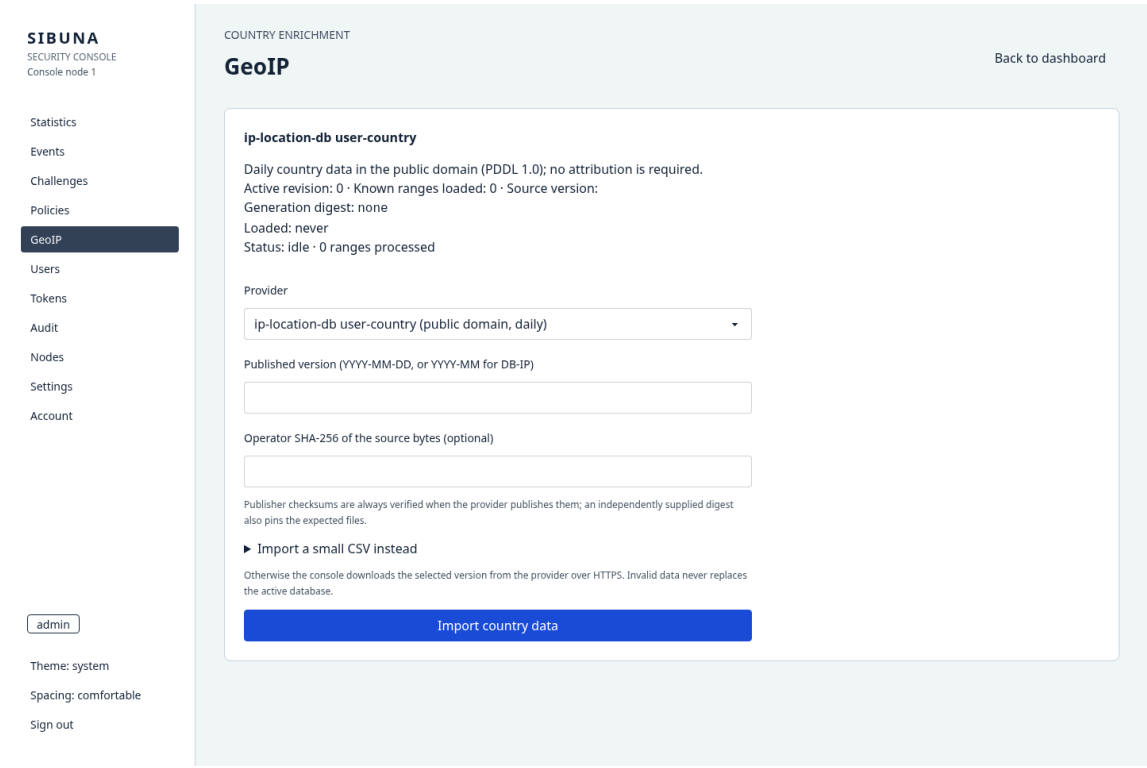


図 20 GeoIP。プロバイダー、ライセンス、有効な世代のダイジェスト、読み込み時刻、インポートのフォームです。ここではまだどの世代も読み込まれていません。失敗したインポートが有効な世代を置き換えることはありません。

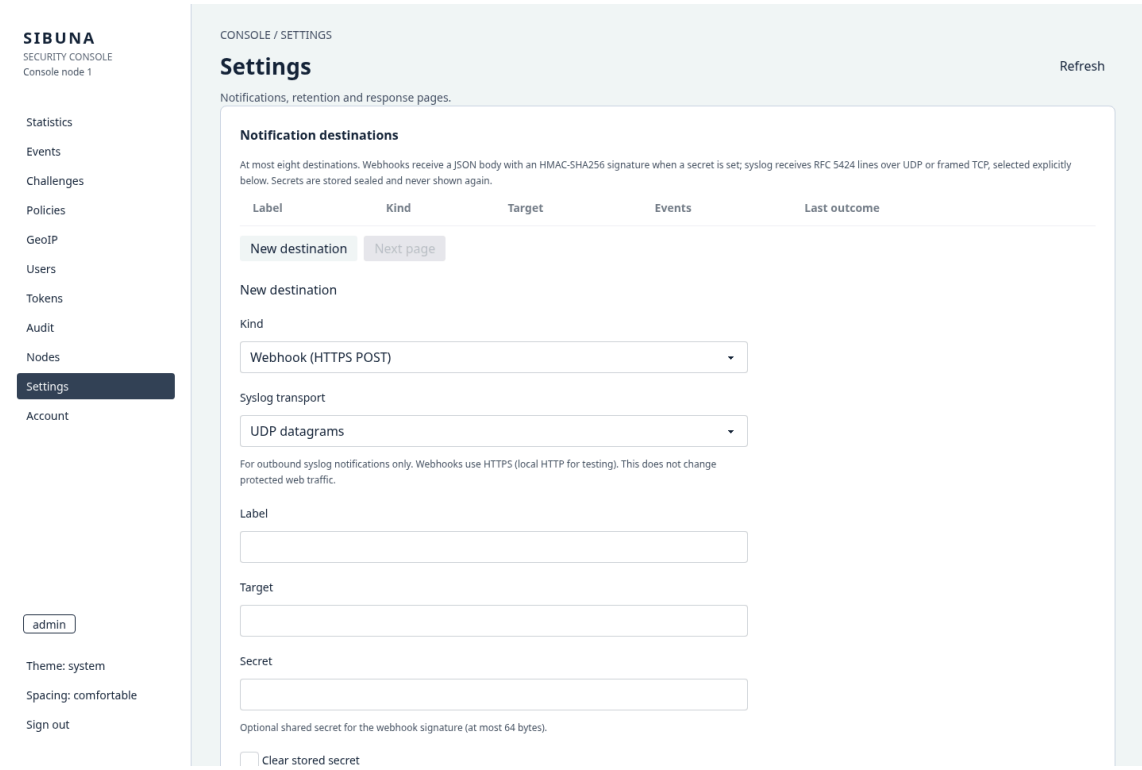


図 21 設定（Settings）。通知先はまだなく、新しい通知先のフォームがあります。ページのさらに下には、拒否の急増のしきい値、窓ごとに明示的な確認を伴う保持期間、応答ページのテンプレート、About があります。

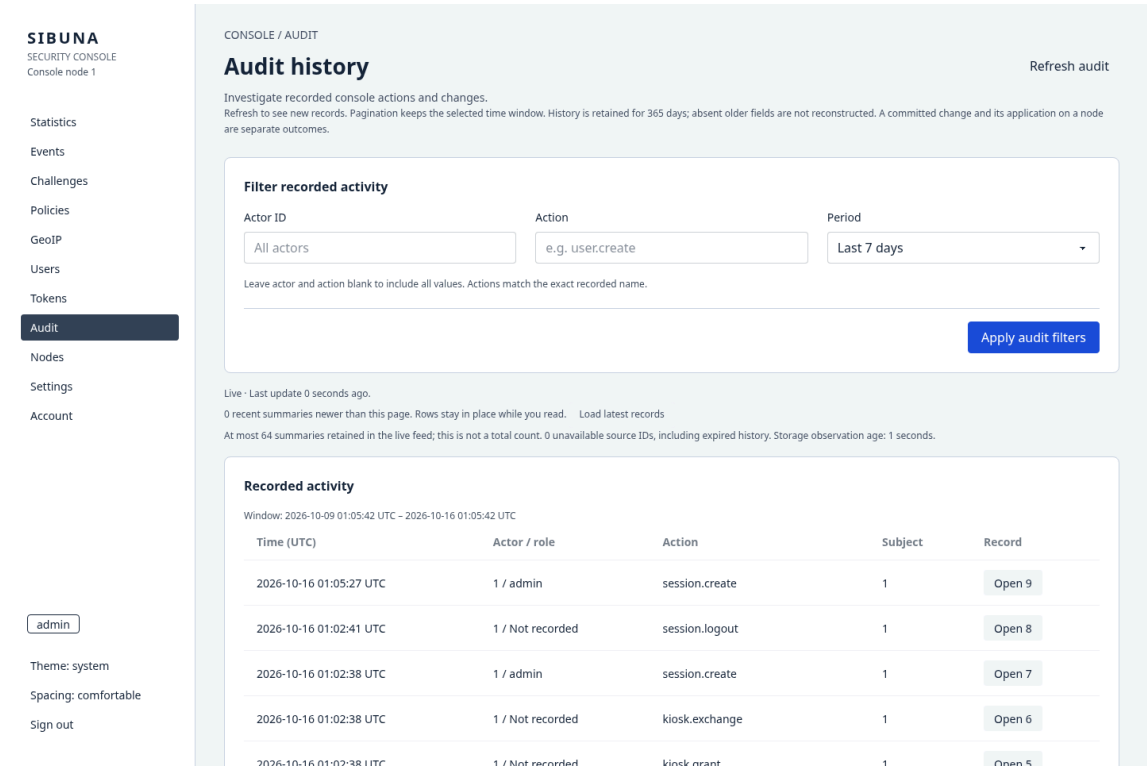


図 22 監査（Audit）。追記専用の履歴で、操作者とロール、操作、対象、伏せ字にした変更前後の要約を含む詳細表示があります。拒否されたサインインも記録されます。管理上の変更と認証の行はすべて、資格情報を提示したクライアントのアドレスを記録します。システムが自ら書く行では、それを記録なしと表示します。ポリシーのレコードには *Revert this change* があり、確認の後で、そのリビジョンの前に記録された文書を、監査付きの新しいリビジョンとして復元します。ルールセットがその後も変わっている場合は拒否されます。

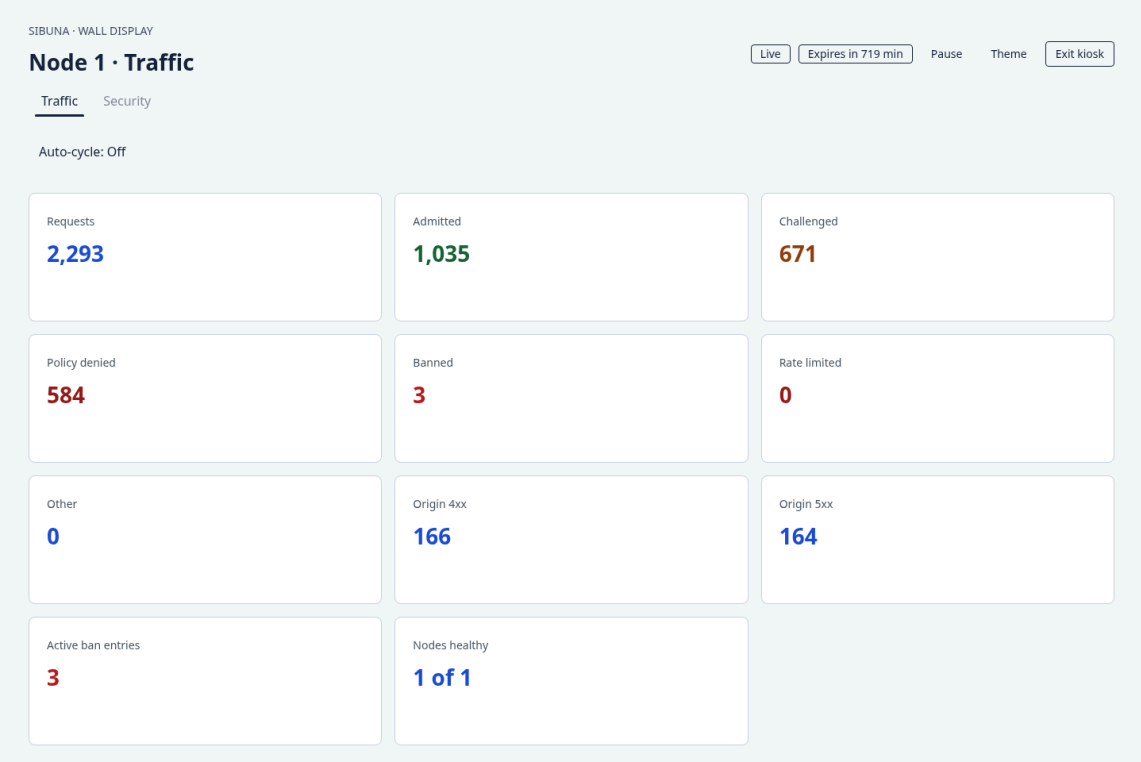


図 23 ウォールディスプレイ。キオスクのセッションは、ナビゲーションなしで Traffic と Security の表示だけを示し、いつ失効するかを示します。

任意の`--console-location <latitude,longitude>`は、このノードの位置を宣言します。たとえばシンガポールのデプロイなら `1.3521,103.8198` です。地球儀は最初にそこを中心に表示し、回転させた後は Center Sibuna でその地点に戻ります。国ごとの活動は、ノードに向かってアニメーションする大円の弧で表され、裏側の半球や平面地図の日付変更線では切り取られます。国の位置は代表的な重心で、矢印は個々の接続ではなく、観測した 60 秒の窓を表します。サーバーの位置を設定しなければ、明示的に不明のままです。設定したマーカーはテレメトリーが途絶えても見えたままで、古いトラフィックはアニメーションしません。

ポリシーのプレビューは、設定したファイルのルールを含む非公開の候補を構築します。保存では期待するリビジョンを比較し、フィールドの比較の確認を求めます。過去への巻き戻しは現在のルールと比較し、新しいリビジョンを作ります。コミット済みのリビジョンとローカルで適用したリビジョンは別のものです。監査の詳細は、上限のある判断の変化と、実際に操作したロールを保持します。照合の値は伏せ字にされ、文脈が欠けた古いレコードは明示的に欠落のままです。ドレイン、再開、ローカルの禁止の解除には、コマンドのプレビューが必要です。ノードで別のコマンドが実行中に届いたコマンドは、キューに入れずに `503 CONSOLENODE` で拒否されます。応答が失われた後は、再試行する前に永続的な受領記録を確認してください。意図がコミットされただけでは、実行時の効果が完了したことにはなりません。

リモートで運用するには、HTTPS プロキシを使い、`--console-origin`、`--console-behind-proxy`、明示的な`--console-trusted-proxy` の CIDR、`--console-key-file` を設定します。鍵ファイルは 16 進数 64 文字を含み、所有者のみの権限を持ち、保存した二要素認証の秘密を保護します。チャレン

ジのシードとは別に、再起動をまたいで保持してください。ループバック以外からのアクセスには、設定した HTTPS オリジンとプロキシの許可リストが必須です。Sibuna はコンソールのリスナーでも受信の TLS を終端しません。コンソールが TLS を使うのは外向きの接続だけで、rustls を使います（SID 0012）。
ネイティブの CLI は、同じ認可とストレージの取り決めを通して稼働中のコンソールにアクセスします。資格情報はコマンドラインの値ではなく、非公開のファイルから読みます。

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

所有者のみのパスワードファイルを使い、先にパスワードの設定を済ませ、必要なら認証アプリまたはリカバリーコードのために`--factor-file`を加えます。アカウントと GeoIP の操作では、`--token-file`がユーザー名とパスワードの代わりになります。クライアントは HTTP/1.1 を話し、リテラルのループバックアドレスを除いて HTTPS を要求し、リダイレクトには決して従わず、各リクエストを 20 秒、各応答を 16 KiB に制限します。`geoip status` は有効な世代を読みます。更新では、選んだプロバイダーが公開するファイルを HTTPS でダウンロードし（既定ではパブリックドメインの `user-country` データセット、`--provider dbip --version YYYY-MM` なら DB-IP Lite）、発行者のチェックサム（または`--checksum`で与えたもの）を検証し、`sibuna-geoip` で範囲を上限内で検証し、新しい世代を永続化してローカルで有効にします。失敗した更新は以前の世代を保ちます。CLI のタイムアウト（`--timeout`、既定 1200 秒）は待つのをやめるだけで、提出済みのストレージの作業は取り消しません。DB-IP Lite には CC BY 4.0 の帰属表示が必要で、そのデータが有効な間だけ表示されます。`SIBUNA_GEOIP_DATA=<snapshot>`付きのビルドは、最初の永続的なインポートまで、検証済みのスナップショット（`geoip-snapshot` ツールで書き出したもの）を埋め込みます。コンソールなしのビルドはこれを `GEOIP001` で拒否します。不明なアドレス、サンプルの損失、古いデータは見えたままです。国のデータをインポートしてもトラフィックは生まれず、すでに期限切れになったサンプルに情報が付加されることもありません。
クラスターのメンバーは、共有データベースのメンバーシップ表（`console_nodes`）を通して、起動時、毎分、そして適用中のポリシーまたは制御のリビジョンが変わってから 2 秒以内に自らを通知し、すべてのコンソールの Nodes ページに現れます。リーダー選出はありません。SQLite のノードはロール `single` と、自分自身をリーダーとして報告します。PostgreSQL のノードはロール `unknown` を報告し、リーダーは報告しません。PostgreSQL のフェイルオーバーは Sibuna からは見えないからです（SID 0007）。このコンソールがヘルスチェックすべきデータプレーンのリスナーを持つピアには`--console-probe <node-id>=<http://ip:port>`を、他のコンソールが表示するリンクには`--console-advertise <origin>`を設定します。直接のライブテレメトリーには`--console-peer <node-id>=<https://origin>`（繰り返し指定可、最大 8）と専用の`--console-peer-key-file` を使います。ピアには`--console-behind-proxy` が必要です。両端が互いを列挙し、信頼できる HTTPS のイングレスを使わなければなりません。ピア鍵は 16 進数 64 文字を含む所有者のみのファイルで、コンソール鍵やチャレンジのシードとは独立に用意します。ピアは、`ring` プロバイダーを使う rustls による、認証付きの TLS WebSocket で通信します（SID 0012）。`--console-peer-ca-file` は、非公開の管理用 PKI のための PEM のトラストアンカーを任意で与えます。与えなければ、クライアントはシステムのルートを使います。ホスト名の検証は常に適用され、無効にできません。各管理用オリジンのホスト（DNS 名または IP アドレ

スのリテラル）は、ピアの証明書に名前が記載されていなければなりません。拒否された証明書はピアを `rejected` とし、`peer.rejected` の監査行を一つ書きます。鍵のローテーションと CA ファイルの変更には、調整した再起動が必要です。ピアの証明書は再起動なしで更新できます。Nodes API とその購読は、受信からの経過時間、時計のずれ、起動の変化、サンプリングの損失を報告します。欠けた観測は利用不可のままで、切断された値は古いままです。受信した統計が別のノード自身の寄与になることはありません。ダッシュボードの *Live traffic scope* は、一つのノードを選ぶか、設定したノードを合算します。*Node coverage and locations* は、欠けている、古い、時計がずれているソース、観測時刻、設定した宛先を示します。国の順位は、省かれたソースの行による不確かさを示します。矢印は受信したノードを保持し、欠けたノードの位置をでっち上げることはありません。合算したレートは、寄与するソースのどれかに連続した区間が欠けていれば未観測のままです。一つのノードを選ぶと、保持している秒単位のデータと現在の分のパスの順位を調べられます。リモートのクエリは認証付きのピア接続を使います。クエリの応答は 4 KiB の WebSocket メッセージ一つに収まらなければならないので、配信するコンソールは収まらないページの行を半分にします。欠けたピアは利用不可のままで、履歴のカーソルはノードの再起動をまたげません。分単位の履歴は、独自のノードの選択とノードごとの行を保持します。ナビゲーションとモバイルのヘッダーは、選んだトラフィックのソースとは独立に、配信しているコンソールのノードを示します。ローカルのコマンドはノードごとのままです。

Traffic のタイルは、選んだノードについて、保持している確定済みの分単位のレコードの直近 24 時間で開きます。期間の選択では、1 時間、7 日、90 日、ライブの起動以降の合計も選べます。Yesterday との差分には、選んだすべてのノードから完全に一致する観測範囲が必要です。地球儀とスパークラインは、引き続き別のライブの 60 秒の窓を表します。保持データの走査は完了の 1 分後に更新され、置き換えられるまで以前の値とその経過時間を表示し続けます。ソースや期間を変えると古い範囲は破棄されます。欠けた履歴がゼロになることはありません。

Compare retained traffic は、二つの確定済みの分の窓を並べて表示します。昨日の同じ窓、直前の期間、または二つ目のノードです。期間と終了オフセットの選択で、両方の UTC の範囲を固定します。各操作は片側につき最大 16 個のコンパクトなページを読み、1 ページには 96 個の保存されたレコードがあります。これで、再起動が重ならない 24 時間の窓を 1 バッチで扱えます。Continue はより長い走査のために同じ境界を保ちます。完全な観測範囲、不完全な観測範囲、欠けた履歴は明示されたままです。レートの百分率には、等しい UTC の窓での重ならない完全な区間が必要で、観測したミリ秒で正規化します。参照がゼロで新しいトラフィックがあれば"New"と表示します。主の Traffic の期間は独立したままです。分単位の形式は過去のプロキシのモードを記録しないので、オリジンの応答の比較は、forward-auth でのゼロを観測した応答と解釈するのではなく、利用不可のままにします。

Compare retained path rankings は、二つの確定済みの窓またはノードを選びます。ノード 0 は、退役したメンバーも含めて、保持しているすべてのノードを含みます。一つの操作は最大 16 個の不変のアーカイブを読み、その識別とチェックサムを確認し、表示する行を選ぶ前にすべてのカウンターを統合します。アーカイブは 60 秒の遅着サンプルの窓の後で封印されるので、最新の確定済みの分はまだ保留中のことがあります。Continue は元の窓とカーソルを保ちます。境界が表すのはサンプリングしたパスの接頭辞であり、正確なリクエストの合計ではありません。ページには、部分的な走査、切り詰め、保持期間の境界、報告されたキュー

の損失を示します。アーカイブが欠けていても、トラフィックがゼロであることや観測範囲が完全であることの根拠にはなりません。

Security は直近 24 時間で開き、選んだ期間を 60 個の等しい時間区間に分けます。その集計は、作業量の上限の下でストレージスレッドが実行します。SQLite では、上限は `--console-query-steps` 個の仮想マシンのステップ（100,000 から 50,000,000、既定 400 万）です。`--console` とともに設定してください。無効な値や繰り返した値は拒否されます。PostgreSQL では、上限は読み込みのトランザクションで設定する 2,000 ms のステートメントタイムアウトです（SID 0013）。`--console-query-steps` を変えても変わりません。上限を使い果たしたクエリは、より狭い期間を求めます。ステップ数が抑えるのはクエリの作業量であり、経過時間ではありません。グラフは目盛りを共有します。Trend の値を展開すると、UTC での区間の開始時刻と、グループ化した正確なカウントが表示されます。これらは保持している検出なので、インシデントの観測範囲が欠けていても、攻撃がなかったと解釈することはできません。

ウォールディスプレイはキオスクのセッションを使います。Account で、運用者がディスプレイに名前を付け、Create display code を選びます。ディスプレイは 10 分以内に一回限りのコードをサインインのフォームに貼り付けます。そのセッションは読み取り専用で、統計に限られ、12 時間以内に失効します。アカウントのページは、移動したときや非表示になったときに表示したコードを消去します。非表示にしても未使用の許可は取り消されず、交換の期限は引き続き適用されます。コードを決して URL に含めてはいけません。ディスプレイは Traffic と Security を共有します。Security は、インシデントのアドレスやペイロードの証拠を含めずに、モジュールの集計の傾向とリクエストの結果を示します。自動の切り替えは任意で、最初は無効であり、動きを減らす設定、古いデータ、Pause のいずれかで中断します。

通知先（拒否の急増、禁止、到達できないメンバー、リーダーの変更についての署名付き Webhook と syslog）は Settings にあります。Webhook の秘密には `--console-key-file` が必要です。保存した通知先の意味を保つためにリーダーの変更の種類は定義されたままですが、リーダー選出がないので、どのノードもこれを発生させません。各通知先は独自のクールダウンと、3 回の再試行の予算を持ちます。配信の監査レコードは結果を示します。完了したキューの履歴は 7 日と 4,096 件のイベントに制限されます。Webhook には安定した `Idempotency-Key` が含まれるので、受信側は、ネットワークでの完了が不確かな場合でも、繰り返しの効果を抑えられます。Webhook は HTTPS で送られ、平文の HTTP はリテラルのループバックアドレスに対してだけ使われます。応答のボディーは 64 KiB まで読みます。syslog の UDP/TCP の選択は外向きの通知にだけ適用され、保護する Web トラフィックとは独立です。手動のテストは意図と完了を Audit に記録し、通知先の結果を更新します。監査上の完了が確認されていない場合は、運用者が再試行する前にそのことを明示します。

Settings では保持期間も管理します。分単位の履歴は 1 から 90 日、順位は 1 から 7 日、インシデントは 1 から 30 日、監査は 1 から 365 日です。上限の値が既定値で、順位は 512 MiB の上限を保ちます。保持期間の値を保存するには確認が必要です。後始末で、古いレコードを上限のあるバッチで永久に削除するからです。後で値を増やしても、削除した履歴は戻りません。古い編集は拒否され、成功した変更は Audit に現れます。

応答ページ（チャレンジ、拒否、レート制限、禁止、過負荷）は、Settings で、固定のプレースホルダーを持つ上限のある HTML として編集できます。下書きはサンドボックス化したタブでプレビューでき、保存したページは次のポリシースナップショットから配信されます。

Policies ページでのポリシーのワークフロー：管理するルールの並べ替え、保持している検査の検出に対する下書きの再生、IP グループと、有効な GeoIP の世代に固定した国のブロッ

クの管理、管理するセットのエクスポートとアトミックなインポート（`sibuna console policies export` と `sibuna console policies import --file <set.json>` も使えます）。インポートは 1 から 128 個の文書を運ぶ一つのリクエストで、各文書は最大 4 KiB、全体で最大 256 KiB です。約 96 KiB を超えるセットについては、ページの注記がコマンドラインでのインポートを示します。適用中のルールは、今日記録されたヒット数、1 時間ごとのスパークライン、アクセシブルな表を示します。ヒットとは、宣言的な照合の評価が成功したことです。一致した WEIGH ルールと最初の終端の一致が数えられます。それより前の検査やレピュテーションによる判断で、評価されないことがあります。非公開のテストはこれらのカウンターを増やしません。*Compare rule hits* は、記録された一つのノードについて二つの確定済みの期間を固定します。リビジョンの履歴には *Compare hits around this edit* があり、編集した分を除き、選んだ長さまでの等しい利用可能な期間を使います。任意の適用リビジョンのフィルターで、関係のない世代を比較から除けます。起動、切り替え、欠けた書き込み、保持期間によって、観測範囲の欠落が目に見える形で残ります。百分率には両側で完全な観測範囲が必要です。ルールの観測と 1 時間ごと・1 日ごとの要約は、分単位の保持期間の設定に従います。編集の前後で観測された変化は比較であり、編集がトラフィックの変化を引き起こした証拠ではありません。

新しい GeoIP の世代をインポートしても、国から導いた既存のレピュテーションの行は自動的には更新されません。国の操作をプレビューして、追加、保持、削除される接頭辞を比べ、適用する前にレビューした差分をページ単位で確認してください。置き換えでは、その国が所有する古い行を削除し、独立に管理している接頭辞は保ちます。世代やポリシーのリビジョンが変わった場合は新たなプレビューが必要で、重なる独立した編集は拒否されます。Events ページには、保持している WAF の検出とハニーポットのインシデントがあります。完全なアクセスログではありません。GeoIP を読み込んでいると、ストレージスレッドはインシデントのバッチのトランザクションの中で、各インシデントの国と世代を記録します。これは永続化の時点での帰属であり、リクエスト時の位置を再構成したものではありません。後のインポートは記録済みの対応付けを変えません。国のフィルターは、大文字の 2 文字のコード、読み込んだ世代にないアドレスを表す `unknown`、対応付けのデータを持たないインシデントを表す `not_recorded` を受け付けます。ソースのグループは、国の混在や観測範囲を明示的に報告します。地球儀の *View events* 操作は、選んだノードを保ったまま、その国の直近 1 時間の保持しているインシデントを開きます。Statistics ページの *Security* 表示は、ライブのレート制限、チャレンジ、禁止の割合と、1 時間、1 日、7 日、30 日にわたる保持している検査とハニーポットの検出を組み合わせます。カテゴリー、ソース、パスのリンクは、同じノードと固定した時間の境界で、通常のインシデントのワークフローを開きます。インシデントのフィルターを適用すると、新しい期間が始まります。検出には監査レコードが含まれ、遮断したリクエストの合計とは別のものです。レピュテーションとルールのヒットの帰属がない場合は、記録なしと表示します。イベントと監査のフィルターは、ラベルをその入力欄と一緒に置き、Apply 操作を別に配置し、列を利用できる内容の幅に合わせます。認証付きの WebSocket 一つがナビゲーションをまたいで存続し、統計、インシデントの要約、ノードの状態、ポリシーのリビジョン、チャレンジ、監査の要約を運びます。新しいインシデントと監査のレコードは *Load latest records* の後ろで待つので、読んでいる間に表が動くことはありません。ポリシーの更新は、開いている下書きを置き換えずに、現在のコミット済みと適用中のリビジョンを示します。フローのカウンターはライブで更新されます。既定以外のチャレンジの計時区分を選んだ場合は、明示的に更新してください。過去のクエリ、詳細の読み込み、変更は HTTP リクエストのままです。

ページのフラグメントはブックマークでき、ブラウザーの「戻る」と「進む」で認証済みのページを開き直せます。サイドバーは、このブラウザーで選んだテーマと表示間隔も記憶します。既定のテーマは System です。SID 0007 は Committed です。コンソールは、ヘッドレス Chromium でゴールデンフィクスチャとブラウザーのワークフローに合格しています（`cargo xtask browser console`、SID 0010 の criterion 5）。エンドツーエンドのスイートは `cmd/sibuna/tests/console_*.rs` で、その PostgreSQL クラスターのシナリオは `SIBUNA_TEST_DATABASE_URL` を設定すると実行されます。SID 0010 の記録によれば、コンソールを組み込んだうえで無効にした場合、ストレージのみの場合、有効にしてアイドル状態の場合のいずれでも、製品の 16 個のワークロードは、コンソールなしでビルドしたデーモンとの差が約 1% 以内に収まり、これはノイズの範囲内です。負荷の下で八つのダッシュボードを動かす SID 0007 の正式な分離の判定は、まだ測定していません（第 VIII 部）。WebAssembly アプリケーション全体には非圧縮で 768 KiB という上限があり、デーモンのビルドスクリプトがビルドを失敗させることでこれを強制します。また、640 KiB というレビューのしきい値があります。1.0.0-beta.3 のモジュールは 724,330 バイト（gzip で 272,568 バイト）で、レビューのしきい値を超えており、SID 0007 はこれをレビュー対象として記録しています。これらはコンソールのワークフローに余地を残すためのプロジェクトの上限であり、ブラウザーでの読み込みと応答性の測定の代わりにはなりません。モジュールの線形メモリは、宣言した最大値の 16 MiB まで必要に応じて伸びます。

デプロイの構成

> **この章の学習目標**　自律的なリバースプロキシと、forward-auth の検証器を、Nginx または Caddy の背後にデプロイします。

リバースプロキシ

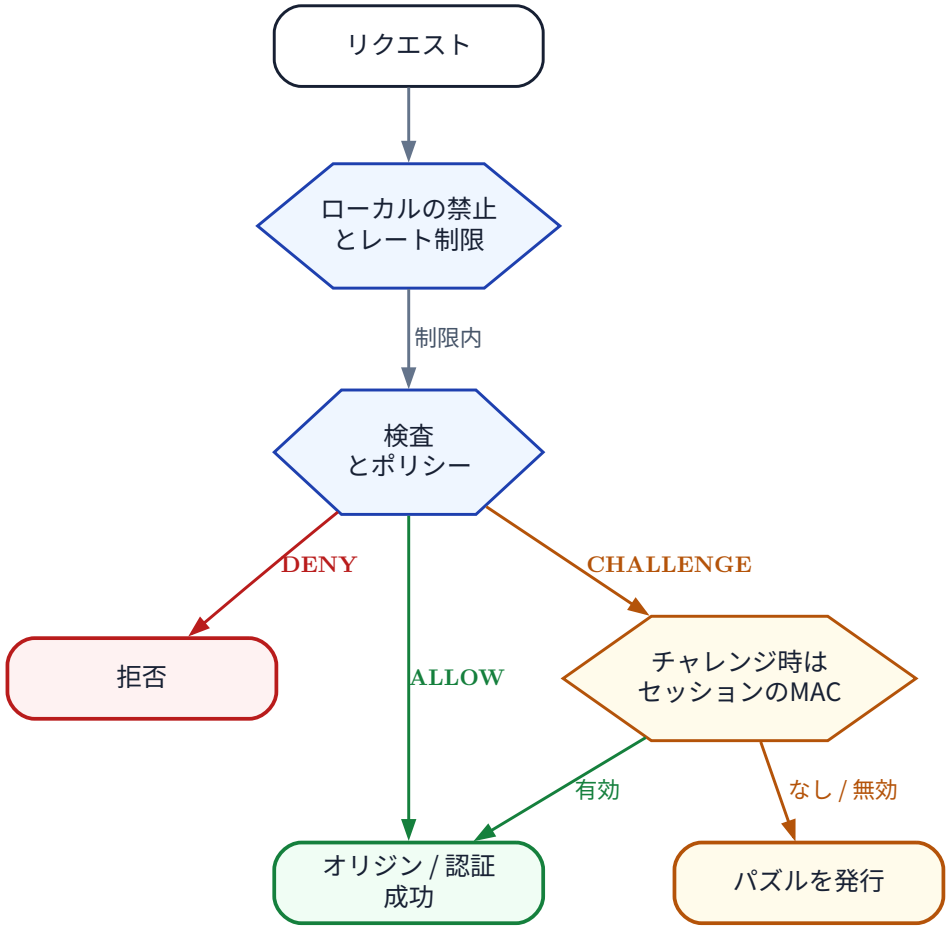


図 24 Shield の運用構成でのリクエストのルーティング

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

許可されたリクエストは、`X-Forwarded-For`、`X-Forwarded-Proto`、`X-Real-IP`、`X-Sibuna-Status`、`X-Sibuna-Rule` ヘッダー付きでオリジンに届きます。ホップごとのヘッダー、Sibuna 自身が書くフィールド、別形式の転送フィールド（`Forwarded`、`X-Forwarded-Host`、`X-Forwarded-Port`）は、クライアントがその名前をアンダースコアで綴った場合（`X_Forwarded_For`）も含めて取り除かれます。`X-Real-IP` は、リクエストの判定に使ったクライアントです。`X-Forwarded-For` はソケットの相手だけで、クライアントが書いた項目は決して含みません。転送された識別情報を信頼する場合は、受信した連鎖の後ろにソケットの相手を続けたものです。したがってオリジンは、クライアントを `X-Real-IP` から、または `X-Forwarded-For` の右側から読まなければならず、左端の項目から読んではなりません。オリジンが受け取るパスは、ポリシーが判定した正規形です（SID 0009）。正規形を持たないパスには `400` で応答します。

HTTP/1.1 の WebSocket のアップグレードは、オリジンのハンドシェイクを受け入れる前に、許可とポリシーの確認を通ります。サブプロトコルと拡張の交渉はエンドツーエンドのままです。二つのスレッドが方向ごとに一つずつバイトを中継し、それぞれが一つの固定バッファーを持ち、先読みしたバイトとハーフクローズも扱います。したがってアップグレードした接続は二つのスレッドを持ち、すべての接続がアップグレードされると、接続スレッドの数は `--max-connections` の 2 倍に達し得ます。アップグレードしたソケットは通常の接続数の上限とシャットダウンの登録簿の中にとどまりますが、HTTP のアイドル期限とは独立に `--websocket-idle-timeout`（既定 300 秒）を使います。どちらの方向のトラフィックもこの上限を更新しますが、期間全体で 1 バイトも受け取らない相手がいれば、もう一方の方向がどれほど忙しくても中継は終わります。ハンドシェイク後のフレームは WAF の検査の入力になりません。HTTPS/WSS では、Sibuna の非公開の HTTP/1.1 リスナーの前に、TLS を終端するイングレスを置きます。Sibuna 自身はブラウザーの TLS を終端しません。

イングレスはブラウザーと HTTP/2 を交渉し、Sibuna には HTTP/1.1 で転送できます。Sibuna でのネイティブの HTTP/2 のサポートは、後の更新に先送りしています。この構成は、ネイティブの gRPC のようなアプリケーションに対して、エンドツーエンドの HTTP/2 の意味論を提供しません。

Content-Length のリクエストボディーは、バイトと MIME ヘッダーを保ったまま固定バッファーを通してストリームされます。マルチパートのアップロードでは、境界、繰り返されたフィールド名、ファイル名、パートの型が保たれます。WAF が調べるのはボディーの先頭 8 KiB までです。マルチパートのメタデータとファイル以外のフィールドはテキストの検査の入力のままで、ファイルのペイロードは中身を見ません。`filename` が空のパートは普通のフィールドです。認識したバイナリのトップレベルの MIME タイプ（SVG を除く画像、音声、動画、PDF、ZIP、gzip、7z、protobuf、octet-stream）のボディーの中身を見ないのは、その先頭部分もバイナリに見える場合、つまり 16 バイトのうち少なくとも 1 バイトが、タブ、改行、垂直タブ、改ページ、復帰以外の制御バイトである場合だけです。そうでなければテキストとして検査します（SID 0004）。JSON、XML、SVG、URL エンコードのフォーム、不明な型はテキストの検査を保ちます。あいまいな、または不正な形式の MIME のメタデータは、テキストの検査に戻ります。マルチパートのパースは、同じ先頭部分の中で 32 パート、パートあたり 2 KiB のヘッダーに制限され、区切りの後の RFC 2046 の転送用の詰め物を受け付けます。

これはアップロードとの互換性であり、ファイルの検証やマルウェアの走査ではありません。バックエンドは、クライアントの Content-Type の宣言を信頼するのではなく、受け付けるメディアタイプを強制しなければなりません。検査する先頭部分より後のフィールドは、大きなアップロードファイルの後にあるものも含めて検査されません。圧縮されたペイロードは、検査のために展開されません。アップロードも、許可、パス、クエリ、ヘッダーの確認を通り、接続の期限の対象のままです。チャンク形式のリクエストボディーは、検査の前に接続バッファーの中でデコードされます（SID 0009）。バッファーの中で終わるボディーは、Content-Length 付きでバックエンドに届きます。それより長いものは Sibuna が読み込みごとに一つのチャンクとしてチャンク化し直すので、バックエンドがクライアントのチャンクサイズ、拡張、トレーラーを見ることはありません。単独の CR や LF、サイズの前後の空白、不正な形式の拡張を含むチャンク行や、4 KiB を超えるチャンク行は 400 で拒否されます。16 KiB を超えるトレーラー部も同様です。他のコーディングを指定した `Transfer-Encoding` には 501 を返します。チャンク形式のリクエストをパースできないバックエンドでも、約 44

KiB 未満のボディーには Content-Length が届きます。リクエストボディーを運んだオリジンへの接続は、そのやり取りの後で閉じられ、プールには戻りません。100-continue を待つクライアントは、ボディーを送る前にローカルでそれを受け取ります。サポートしない expectation には 417 を返します。Expect ヘッダーはバックエンドに転送する前に消費されます。

イングレスの背後での forward auth

--mode forward_auth では、デーモンはイングレスのサブリクエストに、200（と監査用のヘッダー X-Sibuna-Status、X-Sibuna-Rule、X-Sibuna-Rule-Hash）、チャレンジには 401、拒否には 403、レート制限時には 429 で応答します。イングレスはクライアントのアドレスと元の URL を転送しなければなりません。forward-auth モードは既定でそれらを信頼します。イングレスだけが接続できるように、このリスナーは非公開のアドレスにバインドしてください。X-Forwarded-Uri（Caddy）または X-Original-URI（Nginx）と X-Forwarded-Method によって、ポリシーの評価のためにアプリケーションのリクエストを復元します。元のパスは、リクエスト行のパスと同じように正規形に変換されます。重複した、または矛盾する元の URL のフィールドや、origin-form でない元の URI は 400 で拒否されます。デーモンの内部ルートは常に実際のリクエストの URI を使います。forward-auth は、イングレスが省いたボディーを検査しません。

```
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/*の名前空間（中間ページ、チャレンジ、検証、ソルバーの資源）は、どちらのレシピでもデーモンに直接届かなければなりません。nginx のエラーページは元の URI とメソッドを渡すので、中間ページは認可が決めた要件を運びます。それらがなくてもページは動きますが、発行はブラウザーが報告する URL の評価に戻ります。Caddy は元の URI とメソッドのメタデータを自ら与えます。排他的な handle ブロックによって、チャレンジと検証のルートが forward-auth の事前確認に入らないようにします。Nginx では、アプリケーションの WebSocket のために、上に示した明示的な Upgrade/Connection ヘッダーが必要です。Nginx のエラーハンドラーは、二度目の認可判断をしたりアップロードのボディーを消費したりせずに、内部のチャレンジのルートを描画します。その認証モジュールが直接受け付けるのは 2xx、401、403 だけなので、例ではレート制限による認証エラーを Retry-After 付きの 429 に戻します。それ以外の認可の失敗は 503 で閉じたままです。Nginx は独自の 403 のボディーを返しますが、Caddy は Sibuna の拒否ページを転送します。どちらのレシピも、受信した Sibuna の監査用ヘッダーを実際の認可の応答で置き換えます。通常の TLS 証明書とアプリケーショ

ンのアップロードや期限の制限はイングレスで設定してください。例が示すのはルーティングの論理です。リポジトリでは、これらのレシピを実際のイングレスに対してまだテストしていません。レシピが依拠する転送メタデータの規則は、sibuna-http の単体テストで確かめています。

イングレスから受け取る JA4 フィンガープリント

JA4 フィンガープリントは、クライアントが TLS 接続をどのように開くかを要約したものです。これはクライアントの最初のハンドシェイクメッセージである ClientHello から計算します。クライアントの TLS ライブラリは、HTTP のバイトを一つも送らないうちにこのメッセージを確定するので、クライアントは HTTP リクエストの中からフィンガープリントを変えられません。ブラウザー、ライブラリ、ボットはこの値が異なります。そのため JA4 は、どのクライアントも好きなように書ける User-Agent の値よりも強いシグナルになります（SID 0014）。JA4 は FoxIO が定義しました。フィンガープリントは 36 文字で、_でつないだ三つの部分から成ります。

```
t13d1516h2_8daaf6152771_02713d6af862
^^^^^^^^^^ ^^^^^^^^^^^^ ^^^^^^^^^^^^
a          b            c

a  プロトコル（t TCP、q QUIC、d DTLS）、TLSバージョン（13はTLS 1.3）、
   SNIありならd、なしならi、暗号スイートの数（2桁）、
   拡張の数（2桁）、ALPNコード（h2はHTTP/2）
b  ソートした暗号スイートのハッシュ（小文字の16進数12桁）
c  ソートした拡張と署名アルゴリズムのハッシュ
   （小文字の16進数12桁）
```

ClientHello を見るのは TLS を終端するコンポーネントだけなので、JA4 を計算できるのもそれだけです。Sibuna は TLS を終端しません。イングレスの背後で平文の HTTP/1.1 を提供し、ブラウザーとはイングレスが TLS、HTTP/2、HTTP/3 で通信します。すでに JA4 を計算するイングレスもいくつかあります。そこで、イングレスにフィンガープリントを計算させてリクエストヘッダーで送らせ、Sibuna はそこから読み取ります。

--ja4-header <name>がそのヘッダーを指定します。既定では無効で、既定の名前もありません。イングレスが使うヘッダーの名前を常に指定します。Sibuna は名前を小文字で保存し、大文字小文字を区別せずに照合します。自身が書き込む、または取り除くフィールドの名前（X-Forwarded-*、X-Real-IP、X-Sibuna-*、Forwarded。-の代わりに_を使ったものも含む）は拒否します。このオプションを指定すると、起動時のバナーに JA4: from cf-ja4 (trusted)のような行を表示します。

このヘッダーは、転送されたフィールドを信頼する場合にだけ読みます。これは X-Forwarded-For を左右するのと同じ判断で、--trust-forwarded を指定するか、--no-trust-forwarded なしの forward-auth モードです。信頼しない相手からのヘッダーは無視します。信頼する場合、その名前のフィールドがちょうど一つあり、その値が JA4 の形式であれば、リクエストはフィンガープリントを持ちます。値が不正な形式であるか、フィールドが繰り返されていれば、リクエストはフィンガープリントを持ちません。これを二つのカウンターが観測します。sibuna_ja4_present_total はフィンガープリントを持つリクエストを数え、sibuna_ja4_invalid_total はヘッダーが不正な形式であるか繰り返されていたリクエストを数えます。信頼しない相手からのリクエストは、どちらのカウンターも増やしません。

リバースプロキシモードでは、Sibuna がヘッダーを信頼する場合にだけ、オリジンはそのヘッダーを受け取ります。信頼しない場合、Sibuna はオリジンに送るリクエストから、その名前のフィールドをすべて取り除きます。クライアントが名前の-を_で綴った場合も同じです。したがってクライアントは、その名前で偽造したフィンガープリントをオリジンに渡せません。信頼する場合、フィールドはその綴りのまま変更されずに通過し、_を使った別名はやはり取り除かれます。`--ja4-header` がなければ、その名前のフィールドは普通のフィールドです。

> **ヘッダーはイングレスが管理しなければならない**　フィンガープリントは、それを書いたイングレスと同じ程度にしか信頼できません。Sibuna は、イングレスが計算した値とクライアントが送った値を区別できません。したがってイングレスは、Sibuna に届くすべてのルートで、クライアントが与えたヘッダーを上書きするか取り除かなければなりません。`X-Forwarded-For` の場合と同じく、Sibuna のリスナーはイングレスからの接続だけを受け付けなければなりません。

Cloudflare。マネージド変換"Add bot protection headers"は、オリジンに送るリクエストに、`cf-bot-score`、`cf-verified-bot`、`cf-ja3-hash` とともに `cf-ja4` を加えます。これには Bot Management 付きの Enterprise プランが必要です。Cloudflare と Sibuna の間にあるイングレスは、このフィールドを変更せずに渡さなければなりません。Sibuna は次のように起動します。

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

Sibuna の前にあるイングレスは、Cloudflare のアドレスからの接続だけを受け付けなければなりません。そうしないと、クライアントは Cloudflare を迂回して自分の `cf-ja4` を送れてしまいます。

nginx。FoxIO の `ja4-nginx-module` は、nginx の中で JA4 を計算し、`$http_ssl_ja4` として公開します。これには、モジュールの ClientHello 取得パッチを当て、`--with-http_ssl_module` 付きでビルドした nginx 1.26 以降が必要です。`proxy_set_header` はクライアントの同名のフィールドを置き換えるので、Sibuna に届く location ごとに、自らヘッダーを設定します。

```
# リバースプロキシ：Sibunaは --ja4-header x-ja4 --trust-forwarded で動かす
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：Sibunaは --mode forward_auth --ja4-header x-ja4 で動かす
location = /__sibuna_auth {
    internal;
    proxy_pass http://127.0.0.1:8080/;
    proxy_set_header X-JA4 $http_ssl_ja4;
    # ... 上記のforward-authの設定例にある残りのフィールド
}
```

forward-auth のレシピでは、同じ `proxy_set_header X-JA4 $http_ssl_ja4;` の行を `@sibuna_challenge` と `location /__sibuna/` にも置きます。この行を省いた location は、クライアント自身の `X-JA4` を Sibuna に渡し、Sibuna はそれを信頼してしまいます。

Apache Traffic Server。ja4_fingerprint プラグインは、plugin.config の ja4_fingerprint.so という行で有効になります。オプションはありません。このプラグインは、各 TLS 接続のリクエストに、フィンガープリントを持つ ja4 フィールドと、プロキシの名前を持つ x-ja4-via フィールドを加えます。Sibuna は --ja4-header ja4 --trust-forwarded で起動します。プラグインの文書は、クライアントが送った ja4 フィールドを置き換えるかどうかを述べていません。値を信頼する前に、使っているバージョンでこれを確かめるか、プラグインが自身のフィールドを加える前にクライアントのフィールドを取り除かなければなりません。

ルールは ja4 フィールドでフィンガープリントと照合します（第 X 部「ポリシーのスキーマ」）。値は、フィンガープリント全体（等しくなければならない）か、*が続く接頭辞（フィンガープリントがそれで始まらなければならない）のどちらかです。"*"だけなら任意のフィンガープリントに一致します。フィンガープリントのないリクエストが ja4 ルールに一致することはありません。大文字の T13D* のように、どのフィンガープリントも一致し得ないパターンを、ローダーは INVALID POLICY JA4 で拒否します。照合はバイトを比較するだけで、メモリを割り当てません。

```
{ "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" }
] }
```

最初のルールは一つのフィンガープリントを完全一致で拒否します。この値は例であり、既知のスキャナーではありません。次の二つは判断する代わりに、リクエストのスコアに重みを加えます。サーバー名を送らなかった TLS 1.3 のクライアントと、User-Agent で分かるヘッドレスブラウザーです。後のルールが先に判断しなければ、両方のシグナルを示すリクエストはスコアが 45 になって既定の拒否のしきい値 40 に達し、一方だけを示すリクエストにはチャレンジします。最後のルールは、チェックアウトのパスで TCP 上の TLS 1.2 を交渉したすべてのクライアントに、接頭辞によってチャレンジします。拒否ではなくチャレンジにするのは、正当な古いクライアントの多くが今も TLS 1.2 を使うからです。ルールはこのように JA4 を他のシグナルと組み合わせるべきです。JA4 の値が示すのは TLS ライブラリとその設定であり、人や機器ではありません。多くのクライアントが一つのライブラリを共有し、特化したライブラリはブラウザーの ClientHello を模倣できます。JA4 は自動化のコストを上げますが、身元を証明するものではありません。同じ理由から、Sibuna は禁止とレピュテーションをアドレスをキーとして保持し、フィンガープリントをキーにすることはありません。

インシデント（WAF による拒否、ハニーポットへのアクセス、監査の検出）は、そのリクエストがフィンガープリントを持っていれば、それを列 security_incidents.ja4 に記録します。コンソールはインシデントの証拠の中で、これを"JA4 fingerprint"として、値または"Not recorded"とともに表示します。Events のフィルターはフィンガープリントまたは接頭辞を受け付けます。管理するルールのエディターには JA4 フィールドがあり、policies.ja4_pattern に保存されます。ポリシーのテストフォームはフィンガープリントを受け取ります。

宣言的なポリシー

この章の学習目標 ルール、重み、しきい値、IP のルール、WAF の切り替えを含むポリシーファイルを書きます。

```
{
  "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" }
  ]
}
```

ローダーは安全側に失敗します。不明なキー、綴りを誤った動作、不正な形式のアドレス、範囲外の数値があるとファイル全体を拒否し、診断はルールとフィールドを示し、デーモンは起動しません。不明なオプションや範囲外の値を含むコマンドラインも、同じように拒否されます。`rules` があれば組み込みの表を置き換えます。`ip_rules` はレピュテーショントライに供給され、数千の接頭辞まで扱えます。`waf: false` はファイルから Gate の運用構成を選びます。スキーマにはさらに二つのキーがあります（第 X 部）。`inspection` は各 WAF のカテゴリーを `disabled`、`audit`、`enforce` のいずれかに設定し、ルールの `limits` オブジェクトはルールに専用のクォータを与えます。ルールの `ja4` フィールドは、信頼するイングレスが転送した JA4 フィンガープリントと、完全一致または*が続く接頭辞で照合します。これには`--ja4-header` が必要です（「イングレスから受け取る JA4 フィンガープリント」を参照）。パスのパターンは正規形のリクエストのパスと照合するので（SID 0003）、パターンはその形で書かなければなりません。

永続ストレージとクラスタリング

この章の学習目標 SQLite または PostgreSQL のストレージを有効にし、SQL で動的なポリシーと禁止を追加し、インシデントのフォレンジックを問い合わせ、キャンペーンのクラスタリングを理解し、一つの PostgreSQL データベースで複数のノードを動かします。

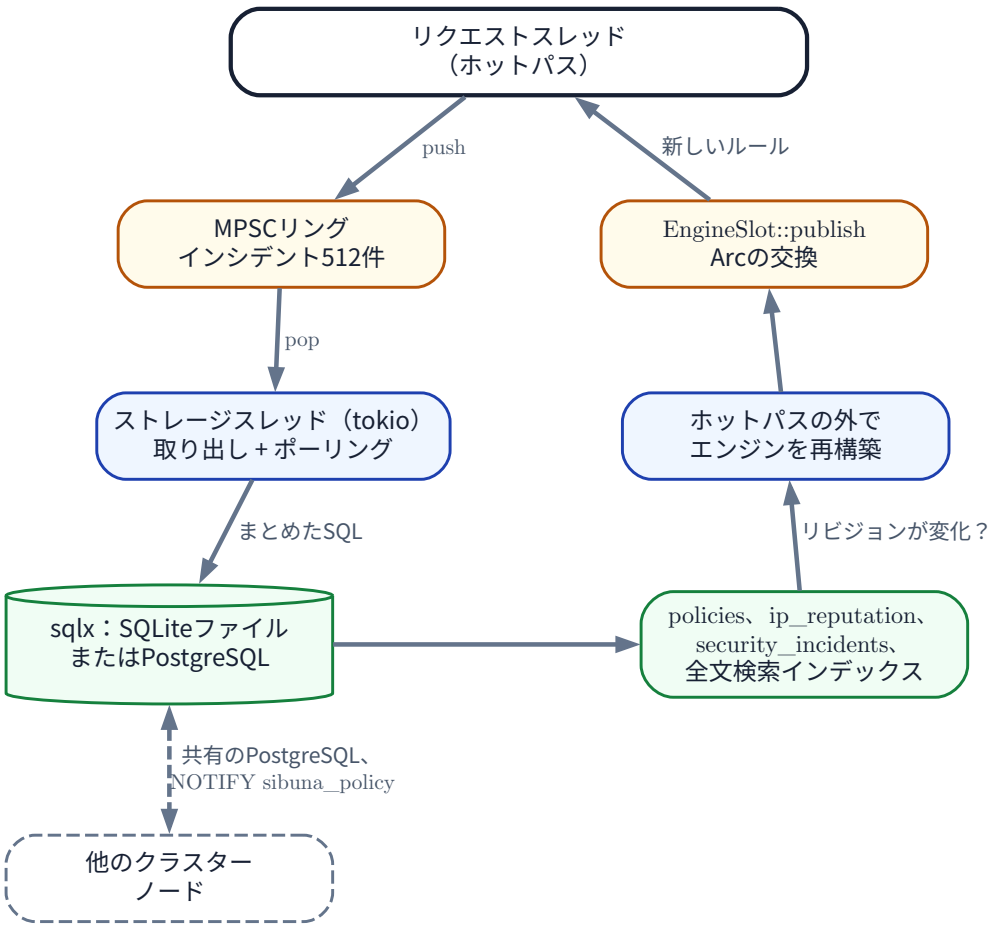


図 25 ストレージのアーキテクチャ：ワーカーがデータベースでブロックすることはない

`--data-dir /var/lib/sibuna` は SQLite ファイル `/var/lib/sibuna/sibuna.db` を開き（WAL モード、`synchronous=NORMAL`、ビジータイムアウト付き）、ストレージスレッドを起動します。`--database-url`（または`--database-url-file`）は、代わりに PostgreSQL データベースに接続します。ストレージスレッドは単一スレッドの tokio ランタイムと小さな接続プールを所有し、SQL を実行する唯一の場所です。起動時、ポリシーのリビジョンが変わったとき（`policies`、`ip_reputation`、`policy_inspection` への書き込み）、レピュテーションの項目が期限切れになったときに、スレッドはポリシーファイルとデータベースからエンジンを再構築して公開します。リクエストが SQL を待つことはありません。再構築では読み込みの前後でリビジョンを読み、その間に書き手がコミットしていれば候補を破棄します。

スキーマ

表 25 中核の表

表	目的とキー
policy_revision	ポリシーのリビジョン、1 行。三つのポリシーの表のトリガーがこれを進めます。
policies	順序付きの動的なルール、キーは id。NULL を許す ja4_pattern はルールの JA4 の条件で、最大 37 バイト。
policy_inspection	カテゴリーごとに保存した WAF のモード、1 行。
ip_reputation	スコア、ヒット数、期限。キーはアドレスまたは接頭辞。
incident_receipts	各ノードの永続的なインシデントの受領記録（最後にコミットしたバッチの次のシーケンス）。キーはノード。
security_incidents	64 成分の埋め込みベクトル（256 バイト）を持つ、上限のある一つのイベントのレコード。キーは node_id << 40 | sequence。NULL を許す ja4 はリクエストの JA4 フィンガープリントで、ちょうど 36 バイト。
incidents_fts	SQLite では、パスとペイロードの正規化したトークンに対する FTS5 の表。PostgreSQL では、同じトークンが GIN 索引付きの security_incidents の tsvector 列 search です。
sibuna_schema	適用したすべてのマイグレーション。キーはコンポーネントとバージョン。

中核のスキーマは crates/storage/src/schema.rs で、方言ごとに一度ずつ書かれています。コンソールがあると、コンポーネント console が、名前が console_ で始まる 33 個の表を加えます（SID 0013、crates/console-storage/src/schema/）。マイグレーションはコンポーネントごとに番号が付き、バイナリにコンパイルされます。中核のマイグレーション 2 で二つの JA4 の列が加わります。それより前に作成したデータベースは、起動時にマイグレーションされます。保留中のマイグレーションはすべて一つのトランザクションで実行され、ノードは、コンポーネントのバージョンが自分の知るものより新しいデータベースを拒否します。省略して印刷したスキーマは、アップグレードのスクリプトではありません。SQLite の表は STRICT で、CHECK 制約が Rust の型を写しているので、デーモンが表現できない行はデータベースが拒否します。列挙値は小文字で保存します。'DENY' を直接書き込むと拒否されます。

運用

sibuna.db には sqlite3 シェルを使い（WAL モードなので、デーモンの実行中も読めます）、クラスターのデータベースには psql を使います。

```
-- 動的なルールを追加する。各ノードは次のティックで再構築する。
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());

-- あるアドレスをクラスター全体で1日間禁止する。
```

```
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());

-- フォレンジック：記録したペイロードを全文検索する。
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;
```

これらの文はSQLite向けに書いています。PostgreSQLでは、enabledはTRUE、時刻はunixepoch()の代わりにCAST(extract(epoch FROM now()) AS BIGINT)、検索はsecurity_incidentsに対するWHERE search @@ CAST('union' AS tsquery)です。pathとoffending_payloadは生のリクエストのバイト（BLOBまたはBYTEA）を保持します。制約に反する直接の書き込みはデータベースが拒否し、ノードには届きません。PostgreSQLでは、リビジョンのトリガーがNOTIFY sibuna_policyも発行するので、他のノードはすぐにティックします。保証となるのは引き続きポーリングです。
−50以下のスコアはトライのdeny接頭辞に、50以上のスコアはallowになり、banned_untilが禁止の期間を区切ります。ハニーポットへのアクセスは自動的に−100のレコードを挿入するので、共有PostgreSQLデータベースでは、一つのノードで作動した罠がすべてのノードでそのアドレスを禁止します。

トランザクションの受領記録が再試行を安全にする

ストレージスレッドは保留中のレコードを最大32件集め、一つのSQLトランザクションを作ります。トランザクションはインシデントを挿入し、全文索引を更新し、ハニーポットによるレピュテーションの変更を適用します。各発行者（ノード）は単調に増加する受領記録、つまりincident_receiptsの自分の行を持ちます。データを変更する文はすべてその受領記録で守られ、トランザクションは最後にそれを進め、決して戻しません。

例題：確認応答が消える 発行者2がシーケンス101から132を用意し、受領記録133で終えるとします。データベースはすべてのレコードと受領記録133をコミットし、その後、確認応答が届く前に接続が切れます。ストレージスレッドは用意したバッチ（識別子、埋め込みベクトル、キャンペーン）を保持して再試行します。受領記録はすでに133なので、守られた書き込みは何もしません。インシデントもハニーポットのヒット数も二重になりません。元のトランザクションがロールバックされていれば、古い受領記録が残り、再試行はすべての書き込みを一度だけ適用します。

コミットに失敗すると、保留中のバッチをメモリに保持し、incident_write_failuresを増やします。ティックでは引き続きポリシーのポーリングを試みます。512レコードのキューが満杯になるとincidents_droppedを数えます。ディスクを待ってHTTPの応答をブロックすることはありません。保留中のバッチは32レコード分の空間を加えます。コミット前にプロセスが死ぬと、キューに入ったデータは失われ得ます。これは上限のある非同期のフォレンジックの経路であり、キューに入れた時点で永続化されるメッセージキューではありません。

演習 9.1. 受領記録の更新をデータのトランザクションの外に移してください。レピュテーションへの効果を重複させるクラッシュの起き方と、インシデントを失わせる起き方を一つずつ構成してください。

キャンペーンのクラスタリング

各インシデントのペイロードは、バイトのトライグラム（数字と大文字小文字を同一視）に対するハッシュトリックで 64 次元の単位ベクトルに埋め込まれ、256 バイトとして `security_incidents.embedding` に保存されます。ストレージスレッドは、全ノードの最新 16,384 個の埋め込みベクトルをメモリ（4 MiB）に保持し、ティックごとにデータベースから更新します。挿入の前に、同じバッチ内の先のレコードも含めて、それらのうち最も近いものを探します。コサイン距離が0.35以下なら、新しいインシデントはそのインシデントの `campaign_id` に加わり、そうでなければ自身の識別子を名前とするキャンペーンを始めます。窓より古いインシデントとは比較しません。類似度はヒューリスティックです。回帰テストの例では関連する SQL のペイロードがまとまりますが、攻撃のどの組も同じキャンペーンに属するという保証ではありません。モデルは実行せず、何も学習しません。

クラスタリング

クラスターとは、一つの PostgreSQL データベースを共有するノードの集合です。`postgres-tls` フィーチャーでビルドし（`cargo build --release -p sibuna --features postgres-tls`。`postgres` も含みます）、各メンバーをデータベースと固有のノード ID で起動します。

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

すべてのメンバーは同じマスターシークレットと、1 から 8,388,607 の一意な`--cluster-node` を使わなければなりません。URL ファイルは `postgres://` の URL を一つ持ち、所有者だけが読めなければなりません。バナーが URL を表示することはありません。書き込みは PostgreSQL のプライマリーに送られます。各ノードのストレージスレッドは、コミットされたリビジョンの変化を（`NOTIFY` またはポーリングで）見て、エンジンを再構築します。
ノードはネットワーク越しに共有データベースに到達するので、その接続を暗号化して検証します。サーバーには自分たちの認証局が署名した証明書を与え、URL でその認証局を指定します。`postgres://sibuna:...@db.internal:5432/sibuna?sslmode=verify-full&sslrootcert=/etc/sibuna/db-ca.pem` です。サーバーでは、`pg_hba.conf` の `hostssl` 行を通してだけ TCP を受け付けます。
起動時、デーモンはデータベースの接続について 1 行表示します。`verify-full` なら、証明書とホスト名を検証したと表示します。それより弱いモードでは、警告とヒントを表示します。`sslmode` がなければモードは `prefer` です。サーバーが TLS を提供する場合にだけ接続は暗号化され、サーバーは検証されません。`postgres-tls` なしでビルドしたデーモンは、`require`、`verify-ca`、`verify-full` を拒否して欠けているフィーチャーを示し、平文に戻ることは決してありません。証明書が拒否されると、起動はすぐに停止します。ヒントは、`sslrootcert` と、URL のホストがサーバーの証明書にある名前であることを確認するよう求めます。`sslrootcert` はシステムのトラストストアに自分たちの認証局を加えますが、システムの認証局を取り除くことはしません（SID 0011、SID 0012）。

Sibuna は独自のデータベースを複製せず、リーダーも選出しません。複製、フェイルオーバー、バックアップは PostgreSQL の役割です（ストリーミングレプリケーション、Patroni、マネージドサービス）。コンソールのノード表示にはリーダーがありません。SQLite のノードにはロール `single` を、PostgreSQL のノードには `unknown` を表示し、`quorum` はノードの直前のデータベースとの往復が成功したことだけを意味します。データベースに到達できなければ、最後に公開したエンジンが配信を続け、インシデントは上限のあるリングに溜まり、満杯になるとカウンターを増やして破棄され、ストレージスレッドはバックオフしながら再試行します。ノードは再接続するまで最後のポリシーを保ちます。Zaxonlite のデータディレクトリ（SID 0005）は読みません。ポリシーは `sibuna console policies export` と `sibuna console policies import` で移し、フォレンジックの履歴は移行しません。

クラスターでのチャレンジのルーティング

チャレンジ鍵は`--cluster-node` に結び付いています。チャレンジの発行と検証は同じメンバーで行ってください。トークンは共有のシードを使い、他のメンバーでも有効です。ローカルのレートのクォータは全体のクォータにはならず、プロセスを再起動すると使用済み集合は消えます。一つのローカルな PostgreSQL に三つのノードを置くと、直接書き込んだポリシーの行は 1 ティック以内に三つすべてに届き、一つのノードでのハニーポットによる禁止は他のノードに届き、インシデントの識別子は衝突しません（`crates/storage/tests/multi_node.rs`、SID 0011 の基準 2。`SIBUNA_TEST_DATABASE_URL` を設定すると実行されます）。

パッケージング

> **この章の学習目標** リリースパッケージをインストールして検証し、Sibuna をサービスとして、コンテナーで、または Kubernetes の下で実行し、バイナリを自分たちでビルドします。

GitHub のどのリリースにも、プラットフォームごとに同じ実行ファイルが含まれます。これはリリースタグから既定のフィーチャー（SQLite ストレージとコンソール）でビルドしたものです。PostgreSQL のクラスターには、自分たちでビルドしたものが必要です（後述）。

プラットフォーム	リリースアセット
Linux x86-64、ARM64	`sibuna-linux-amd64.tar.gz` と `sibuna-linux-arm64.tar.gz` には、どのディストリビューションでも動く静的な musl の実行ファイルが入っています。`sibuna_<version>-1_amd64.deb`、`sibuna_<version>-1_arm64.deb`、`sibuna-<version>-1.x86_64.rpm`、`sibuna-<version>-1.aarch64.rpm` は、それを systemd サービスとともにインストールします。Arch Linux では `sibuna-bin-<version>-1-x86_64.pkg.tar.zst` が同じことをします。
macOS arm64、x86-64	`sibuna-macos-arm64.tar.gz`、`sibuna-macos-amd64.tar.gz`。macOS 12 以降に対応します。
Windows x86-64	`sibuna-windows-amd64.zip` には `sibuna.exe` が入っています。Windows 10 または Server 2016 以降に対応します。

プラットフォーム	リリースアセット
FreeBSD 15.1、OpenBSD 7.9（amd64）	sibuna-<version>-freebsd-15.1-amd64.pkg と sibuna-<version>.tgz。
Homebrew、Kubernetes、コンテナー	sibuna.rb、helm-sibuna-<version>.tgz、そして amd64 と arm64 向けのイメージ ghcr.io/insanai/sibuna:<version>。イメージのダイジェストは IMAGE-DIGEST.txt にあります。
ソース	sibuna-<version>-source.tar.gz。

各アーカイブとパッケージには、LICENSE、NOTICE、LICENSES/（実行ファイルに含まれるクレートを記した THIRD-PARTY-RUST.txt を含む）、README.md、SECURITY.md、sibuna.build.json が入っています。このマニフェストは、バージョン、コミット、Rust のターゲット、コンパイラーのバージョン、フィーチャー、そして実行ファイルとその二つの WebAssembly モジュールの SHA-256 を記録します。リリースワークフローは、何かを公開する前にすべての実行ファイルをそれぞれのプラットフォームで起動し、Debian、Fedora、Arch Linux、FreeBSD、OpenBSD でパッケージのインストール、再インストール、削除を行います。
アセットには署名がありません。インストールする前に、ダウンロードしたものを同じリリースの SHA256SUMS と照合します。

```
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
```

macOS では、ブラウザーでダウンロードしたファイルに隔離属性が付き、これが署名のない実行ファイルを止めます。照合の後、xattr -d com.apple.quarantine sibuna でこの属性を取り除きます。

systemd サービス

Debian、RPM、Arch のパッケージは、/usr/bin/sibuna、ユニット sibuna.service、（systemd-sysusers による）アカウント sibuna、ヘルパー/usr/lib/sibuna/seed.sh、設定ファイル/etc/sibuna/service.env をインストールします。インストールしても何も起動しません。設定ファイルにオリジンを IP リテラルで設定し、サービスを起動します。

```
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
```

サービスは、SIBUNA_HOST と SIBUNA_PORT で別の指定をしない限り、127.0.0.1:8080 で待ち受けます。SIBUNA_OPTIONS は、空白で区切った追加のオプションを渡します。たとえば--mode forward_auth や--console 127.0.0.1:19446 です。起動のたびに、seed.sh は許可シード/var/lib/sibuna/admission.seed（16 進数 64 文字、モード 0600、所有者 sibuna）がなければ作成します。既存のシードを置き換えることはありません。新しいシードはすべてのセッションを終わらせてしまうからです。デーモンは SQLite データベースを、モード 0700 のディレクトリ/var/lib/sibuna に保存し

ます。ユニットは、ケーパビリティなし、読み取り専用のシステム（ProtectSystem=strict）、プライベートな/tmp とデバイス、新しい特権なしで動きます。
アップグレードは設定ファイル、シード、状態を保ち、サービスを再起動しません。バックアップの後で自分たちで再起動します。削除はサービスを停止して無効にしますが、アカウント、シード、設定、状態は残します。これらはホストを退役させるときに自分たちで削除します。
FreeBSD と OpenBSD のパッケージがインストールするのは、/usr/local/bin/sibuna とその文書だけです（pkg add、または OpenBSD のミラー以外から入手したパッケージには pkg_add -D unsigned）。サービス、アカウント、状態は作成しません。デーモンは、専用のシードとデータディレクトリを持つ特権のないユーザーで実行します。

Homebrew

sibuna.rb は、マシンに合ったリリースアーカイブ（macOS または Linux、arm64 または x86-64）をインストールします。ソースからはビルドしません。Homebrew の Rust には wasm32-unknown-unknown 向けの標準ライブラリがなく、埋め込む二つの WebAssembly モジュールのビルドにはそれが必要だからです。アーカイブには、リリースワークフローが確認済みの実行ファイルが入っています。Homebrew は tap からしか formula をインストールしないので、formula をローカルの tap に置きます。

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

この formula はサービスを起動しません。注意書き（caveats）はシードの作り方を示し、テストは同じデータディレクトリでデーモンを 2 回起動します。

コンテナーと Kubernetes

イメージ ghcr.io/insanai/sibuna:<version>は、distroless のベースに静的な実行ファイルを載せたものです。CA 証明書とユーザー 65532 を含み、シェルもパッケージマネージャーもありません。エントリーポイントは sibuna なので、引数はデーモンのオプションです。リリースワークフローはこのイメージを、読み取り専用、ケーパビリティなしで、シードを読み取り専用でマウントし、状態用のボリュームを付けて実行し、1 回再起動します。

```
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 チャート helm-sibuna-<version>.tgz は、そのイメージの Pod を一つ実行します。自分たちで作成する Secret（secret.existingSecret、キー admission.seed）が必要なので、シードが values やテンプレートに現れることはありません。Pod は、Restricted のポッドセキュリティ標準の下で、読み取り専用のルートファイルシステムを持つ UID 65532 として動きます。Recreate でアップグレードするレプリカ一つが、ReadWriteOnce のボリュームの書き手を一つに保ちます。Sibuna はこのボリュームの data/に SQLite の状態を保存し、アンインストールしても

ボリュームクレームは残ります。プローブは/__sibuna/health を呼びます。Service は ClusterIP で、チャートは ingress もコンソールも作成しません。チャートの NetworkPolicy は、networkPolicy.ingressFrom と networkPolicy.egressTo がクライアントとオリジンを許可するまで、すべての通信を拒否します。upstream.host は IP リテラルでなければならず、extraArgs は追加のデーモンオプションを渡します。ワークフローは、Restricted 標準を強制する kind クラスターで、チャートのインストール、アップグレード、アンインストールを行います。

```
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
```

ソースからビルドする

私たちは Rust 1.100 でビルドとテストを行います。サポートする最小バージョンは引き続き Rust 1.94 です。

リリース用ツールチェーンを rustup toolchain install 1.100.0 --profile minimal --target wasm32-unknown-unknown でインストールし、以下のコマンドでは cargo +1.100.0 で選択します。

- cargo build --release -p sibuna は、既定のフィーチャー（SQLite ストレージとコンソール）で target/release/sibuna を生成します。sqlx は SQLite を同梱し、ring（rustls）はビルド時に C コンパイラーを必要とします。バイナリはシステムの C ランタイムを動的にリンクします。ビルドスクリプトは WebAssembly のソルバーとコンソールのインターフェースもビルドするので、rustup target add wasm32-unknown-unknown が必要です。SIBUNA_POW_WASM=<path> と SIBUNA_CONSOLE_WASM=<path>を指定すると、代わりにビルド済みのモジュールを埋め込みます。
- cargo build --release -p sibuna --no-default-features は、外部クレートをまったく使わない、メモリ上のファイアウォールを生成します。この場合、--data-dir と--database-url は起動時に StorageDisabled で、sibuna console は CONSOLEBUILD で拒否されます。
- cargo build --release -p sibuna --features postgres-tls は、データベースへの検証付き TLS を備えた PostgreSQL のクラスターを加えます。OpenSSL は必要ありません。
- リリースの静的な Linux 実行ファイルは、musl-gcc を C コンパイラーとして（CC_x86_64_unknown_linux_musl=musl-gcc）、cargo build --release --locked -p sibuna --target x86_64-unknown-linux-musl（または aarch64-unknown-linux-musl）でビルドしたものです。

リリースプロファイルは fat LTO と一つのコード生成単位を使います。ワークフローは二つの WebAssembly モジュールを一度だけビルドしてすべてのプラットフォームに渡すので、リリースのすべての実行ファイルはバイト単位で同一のモジュールを埋め込みます。依存関係が変わったら、cargo xtask licenses で LICENSES/THIRD-PARTY-RUST.txt を再生成します。

Windows では、デーモンは Ctrl+C、Ctrl+Break、コンソールを閉じるイベントで停止します。Windows サービスとしては動きません。Windows のファイルにはモードビットがないので、Sibuna は他者が読めるかもしれない鍵、認証情報、データベース URL のファイルを拒否できません。これらのファイルは ACL で制限します。コンソールのメモリと CPU のゲージは/proc から読むので、他のシステムでは使えません。

演習 9.2. ステージング用のネットワーク（`10.20.0.0/16`）がチャレンジを回避し、`/admin/*` が 20 作業ビットの逐次作業を要求し、`X-Partner-Key` で識別するパートナーのスクレイパーを 10 秒あたり 30 リクエストで許可するために必要な、ポリシーファイルと `ip_reputation` の行を書いてください。
ヒント：デーモンのフラグは、すべてのリクエストが最初に通るクライアントごとの制限を設定します。ルールは `limits` で専用のクォータを持てます。

不変条件を説明してください。 `policies` に行を加えると、再起動なしに、そしてどのリクエストも待たせることなく効果が現れる理由を、ストレージスレッドとエンジンスロットの観点から運用者に説明してください。

第 X 部

手元リファレンスと診断

エンドポイント、メトリクス、エラーの一覧、ポリシーのスキーマ、ストレージの表を一か所にまとめます。

エンドポイント

この章の学習目標 /__sibuna/配下のすべてのルートと、それぞれが返すものを把握します。

表 27 予約されたルート

エンドポイント	メソッド	機能
/__sibuna/challenge	GET	中間ページ
/__sibuna/challenge.json?need=&path=	GET	状態を持たないチャレンジを発行します。チャレンジを求めた応答の要件チケットがあればそれで決め、なければ報告された URL を正規形で評価します（8 KiB を超えると 414、正規形を持たないパスは 400）。{id, algorithm, difficulty, challenges, expires_at}を返します
/__sibuna/verify	POST	{"challenge_id", "nonce"}または{"challenge_id", "proof"}を受け付けます。Set-Cookie 付きの 200、診断付きの 400、または接続バッファーより大きいボディーには 413 を返します
/__sibuna/wasm/sibuna-pow.wasm	GET	ソルバーのモジュール、14,744 バイト（benchmarks/results/primitives-latest.json の wasm_bytes）。1 時間キャッシュ可能
/__sibuna/worker.js	GET	WASM と JavaScript の証明器を持つ Web Worker。1 時間キャッシュ可能
/__sibuna/honeypot	GET	呼び出し元を--ban-seconds の間禁止し、インシデントを記録します。403 を返します。禁止するのはトップレベルのリクエスト（Sec-Fetch-Dest がないか document で、クロスサイトの Sec-Fetch-Site が

エンドポイント	メソッド	機能
		ないもの）だけです。それ以外には 404 で応答し、数えます
/__sibuna/health	GET	エンジン名、バージョン、プロキシのモード、証明のアルゴリズムを含む JSON の生存状態
/__sibuna/metrics	GET	Prometheus のテキスト形式（バージョン 0.0.4）

/__sibuna/配下のそれ以外のパスには 404 で応答します。発行と検証は別の--challenge-rate-limitの予算を消費し、禁止されたアドレスはどのルートでも 403 で拒否されます。チャレンジの応答は X-Sibuna-Status: CHALLENGE を運びます。許可されたリクエストは上流に X-Sibuna-Status: PASS と X-Sibuna-Rule を運びます（forward-auth の応答では、16 進数の X-Sibuna-Rule-Hash も運びます）。HTML を受け付けないクライアントは、{"error":"challenge_required","challenge":"/__sibuna/challenge.json?need=<ticket>"}付きの 401 を受け取ります。
別のオプトインの管理用リスナーが/console/を配信します。管理者は sibuna init-admin でローカルに初期設定し、最初のサインインで一時パスワードを変更します。/console/api/stats と/console/ws には認証済みのセッション（キオスクのセッションを含む）が必要です。WebSocket は、統計、イベント、ノード、ポリシー、チャレンジ、監査を、上限のあるスナップショット、差分、欠落の回復とともに多重化します。管理用のピアは/console/peer に接続し、API のルートはすべて/console/api/配下にあります。forward-auth モードでは、オリジンのエラーカウンターは観測されません。コンソールのワークフロー、HTTPS でのデプロイ、SID 0007 の現在の制限は第 IX 部に記載しています。

メトリクス

Prometheus のテキスト形式で sibuna_<name>_total として、次の順に公開するカウンターです。

表 28 カウンター

カウンター	増加する条件
incidents_persisted, incident_batches	ストレージスレッドがコミットを確認したレコードとトランザクション
incidents_dropped, incident_write_failures	リングが満杯で拒否されたキューへの追加と、失敗したコミットの試行
requests	パースしたリクエストがディスパッチに達した（予約されたルートを含む）
allowed, denied, challenged	ポリシーの判断が下された（許可、拒否、チャレンジの発行または再発行）
challenges_issued	/__sibuna/challenge.json がチャレンジレコードを発行した

カウンター	増加する条件
solutions_accepted, solutions_rejected	/__sibuna/verify が証明を受理または拒否した
rate_limited	GCRA がリクエストを拒否した（429）：リクエストの上限、チャレンジの予算、ルールのクォータ
banned	禁止されたアドレスを拒否した、またはハニーポットがアドレスを禁止した
bans_issued	禁止を記録した（ハニーポット、またはルールのクォータによる禁止）
proxied, upstream_errors	リクエストをオリジンに中継した、またはオリジンが失敗した（502）
parse_errors	不正な形式のヘッダー部、繰り返された Host、正規形を持たない対象（400）、サポートしない転送コーディング（501）を拒否した
overloaded	--max-connections または--max-connections-per-ip を超えた接続に 503 で応答した
bans_refused	キーが入り得るすべてのスロットに有効な禁止があったため、禁止を記録しなかった
forwarded_invalid	信頼する転送されたクライアントアドレスをパースできなかった。リクエストはソケットの相手で判定した
honeypot_refused	人ではなくページが要求したため、ハニーポットへのアクセスで誰も禁止しなかった
rate_evictions	リミッターへの到着が、別の有効なクライアントのセルを追い出した
ja4_present	--ja4-header を指定し、転送を信頼する場合に、リクエストが JA4 フィンガープリントを持つその名前のヘッダーをちょうど一つ運んだ
ja4_invalid	--ja4-header を指定し、転送を信頼する場合に、ヘッダーが不正な形式であるか繰り返されていた。リクエストはフィンガープリントなしで判定した

保留中のバッチは保持されるので、再試行はレコードを失わずに incident_write_failures を増やすことがあります。一定の間隔での増加を監視してください。合計だけではキューの深さは分かりません。

ステータスコード

表 29 データプレーンのステータスコード

コード	発生条件
200	許可（forward-auth）、中間ページ（チャレンジが必要な HTML のページ遷移）、内部ルート
302	現在のプロトコルでは使いません。検証は `200` で応答し、ページが再読み込みします
400	不正な形式のリクエスト、繰り返された `Host` フィールド、正規形を持たない対象、無効な転送メタデータ、不正な形式のボディー、または Elm 形式の診断付きで拒否された解答
401	HTML を受け付けないクライアント、または forward-auth モードでチャレンジが必要
403	ポリシーまたは WAF による拒否、禁止されたアドレス、ハニーポット
404	`/__sibuna/`配下の不明なパス、またはトップレベルのリクエストでないハニーポットへのアクセス
413	64 KB の接続バッファーより大きい解答のボディー
414	8 KiB を超える、報告されたチャレンジの URL
417	サポートしないリクエストの expectation。`100-continue` はローカルで処理します
429	GCRA の制限を超過。`Retry-After` は秒単位
431	16 KB を超えるリクエストのヘッダー部
501	chunked 以外のコーディングを指定した `Transfer-Encoding`
502	オリジンに到達できない、またはその応答のヘッダー部が不正な形式か 16 KB より大きい
503	接続数の上限（`--max-connections` または`--max-connections-per-ip`）に到達、またはノードがドレイン中。応答の後でソケットを閉じます

エラーの一覧

`INVALID COMMAND LINE` は、デーモンが認識しないオプション、文書化した範囲外の値、無効、欠落、重複したモードの選択のいずれかがあると起動を停止します。このブロックは、オプション、与えられた値、期待される範囲、エラー（`UnknownOption`、`MissingValue`、`InvalidValue`、`InvalidMode`、`MissingMode`、`DuplicateMode`、Zaxonlite のクラスターオプションには `RemovedOption`、互いに排他的な二つのストレージオプションには `ConflictingOptions`、`--cluster-node` のないデータベース URL には `MissingOption`）を示し、ヒントで終わります。データベースの URL は`[redacted]`と表示されます。`--mode reverse_proxy` または`--mode forward_auth` を一度だけ指定してください。`-m` は同等の短いオプションです。拒否されたポリシーファイルは、そのエラーを題にしたブロック（たとえば `INVALID POLICY ACTION`）を、ファイル、エラー名、ルール、フィールド、値とともに表示し

ます。どのフィンガープリントも一致し得ない ja4 のパターンは INVALID POLICY JA4（InvalidJa4）を表示します。そのようなパターンを含むコンソールの管理文書も、同じエラー名で拒否されます。コンソールのオプションは別に確認します。CONSOLE002 は不正な形式または繰り返されたコンソールのオプションを、CONSOLE001 は規則に反するコンソールの設定（ストレージなしでの StorageRequired）を示します。

実装：ErrorKind::explain すべてのドメインのエラーを、境界線、説明、Hint:に対応付けます。crates/core/src/errors.rs（クレート sibuna-core）にあります。

表 30 チャレンジとセッションのエラー

エラー	原因	ヒント
MalformedChallenge	識別子が正しい形式のチャレンジレコードではない	新しいチャレンジを取得し、変更せずに提出する
InvalidChallengeTag	タグが認証できない。このクラスターが発行したものでないか、編集されている	チャレンジは偽造できない。新しいものを要求する
ChallengeExpired	チャレンジの TTL より古い、または未来の時刻で発行されている	新しいチャレンジを要求する
FingerprintMismatch	別のアドレスまたは User-Agent から提出された	取得したクライアントから提出する
DifficultyNotMet	Hashcash の nonce に必要なゼロビットがない	nonce の探索を続ける
InvalidProof	逐次作業の証明が、コミットしたラベルを開示しない	発行された深さと開示数で、証明器を最後まで実行する
WrongSolutionType	PoSW のチャレンジに nonce、または Hashcash に証明	解答のフィールドをアルゴリズムに合わせる
DoubleSpendAttempt	チャレンジはすでに使用済み	チャレンジは一回限り
StoreFull	使用済み集合が満杯	チャレンジの TTL を下げるか、--spent-tags を上げる
InvalidTokenSignature	Cookie のタグまたは署名が一致しない	中間ページで認証し直す
TokenExpired	Cookie の期限切れ	認証し直す
TokenBoundAddressMismatch	別のクライアントの識別情報から提示された Cookie	Cookie は共有できない

コンソールの API は失敗を`{"error":"<CODE>","hint":"<sentence>"}`で返します。hint はインターフェースが表示する文です（`crates/console/src/http/reply.rs`）。

表 31 運用者に関係するコンソールの失敗コード

コード	ステータス	意味
CONSOLE002	400	リクエストが不正な形式、大きすぎる、またはその形式では許可されない
CONSOLE003	503 / 429	ストレージのジョブキューが満杯（32 ジョブ）、またはパスワードの検証器が使用中
CONSOLEQUORUM	503	データベースがリクエストの待ち時間 10 秒以内に応答しなかった。結果は不明で、データプレーンは最後に適用したポリシーで動作を続ける
CONSOLE004	503	サービスが応答できない。このビルドでは利用できない、停止中、または内部の失敗
CONSOLENODE	400 / 409 / 503	ノードへのコマンドが拒否された。別のノードを指している（400）、起動または制御のリビジョンが古い（409）、または別のコマンドの実行中に届いたか完了を記録できなかった（503）。再試行する前に操作の受領記録を確認する
CONSOLEMUTATION	429	一つのセッションから 1 分間に 60 件を超える管理上の変更

ポリシーのスキーマ

```
{
  "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": [                                                  // 最大128個
    {
      "name": "<kebab-case>",                                 // 表示可能な1から128バイト
      "path" | "path_regex": "<pattern>",                     // 最大512バイト
      "user_agent" | "user_agent_regex": "<pattern>",
      "headers" | "headers_regex": { "<Header>": "<pattern>" }, // 最大4個
      "remote_addresses" | "cidrs": ["<cidr>", ...],          // 最大8個、IPv4またはIPv6
      "action": "ALLOW" | "DENY" | "CHALLENGE" | "WEIGH",
      "weight": int,                                          // WEIGHのみ、-1000..1000
      "challenge": { "difficulty": <work bits>, "algorithm": "hashcash" | "posw" },
      "limits": { "rate": int, "window_seconds": int, "ban_seconds": int },
      "ja4": "<fingerprint>" | "<prefix>*"                    // 36バイト、または接頭辞と*
    }
```

```
  ]
}
```

challenge_at と deny_at は-1000 から 1000 の範囲で、deny_at は challenge_at 以上です。bits_step は 1 から 1000 です。challenge ブロックは CHALLENGE ルールにだけ置け、作業ビットは 1 から 64 です。limits はルールに専用のクォータを与えます。rate（1 から 1,000,000）と window_seconds（1 から 86,400）は必須で、ban_seconds（最大 86,400）は任意です。WEIGH ルールは制限を持てません。ja4 は、JA4 フィンガープリント全体（等しくなければならない）か、*が続く接頭辞（フィンガープリントがそれで始まらなければならない）です。*だけなら任意のフィンガープリントに一致し、フィンガープリントのないリクエストはどの ja4 ルールにも一致しません。どのフィンガープリントも一致し得ないパターンは InvalidJa4 で拒否されます（第 IX 部「イングレスから受け取る JA4 フィンガープリント」）。ip_rules のレピュテーショントライは最大 8,192 ノードを保持します。各ルールでは、各条件について一つの綴りだけを使えます。
パターンの文法：.*または*は何にでも一致します。^...$ はパスの完全一致を固定します。末尾の*、/*、.*は接頭辞です。/で始まるパターンはパスの完全一致です。それ以外は大文字小文字を区別しない部分文字列です。パスのパターンは正規形のパス（セグメント間のスラッシュは一つ、ドットセグメントなし、予約されていない文字はエスケープを解除、その他のエスケープは大文字の 16 進数字）と照合します。そのため;、%2F、%5C、\を含むパターンは何にも一致せず、パスの照合は大文字小文字を区別します（SID 0003）。

ストレージのテーブル

policy_revision、policies（最大 37 バイトで NULL を許す ja4_pattern 条件付き）、policy_inspection、ip_reputation、incident_receipts、security_incidents（256 バイトの埋め込みベクトルと、36 バイトで NULL を許す ja4 フィンガープリント付き）、全文索引（SQLite では正規化したトークンに対する FTS5 の incidents_fts、PostgreSQL では GIN 索引付きの tsvector 列 search）、sibuna_schema です。コンソールは 33 個の console_*表を加えます。保存する動作、アルゴリズム、モードの名前は小文字です。インシデントの ID は node_id << 40 | sequence で、調整なしにクラスター全体で一意です。ノード ID は 1 から 8,388,607 です。

ビルドターゲット

表 32 ビルドとリポジトリのコマンド

コマンド	結果
cargo build --release -p sibuna	SQLite ストレージとコンソールを持つデーモン（既定のフィーチャー）
cargo build --release -p sibuna --no-default-features	ストレージもコンソールも外部クレートも持たない、メモリ上のデーモン
cargo build --release -p sibuna --features postgres-tls	共有の PostgreSQL データベースにも検証付きの TLS で接続するデーモン（postgres フィーチャーだけでは TLS なしで接続します）

コマンド	結果
`cargo test`	単体テスト、稼働中のデーモンに対する結合テスト、ストレージとコンソールのテスト（`SIBUNA_TEST_DATABASE_URL` を設定すると PostgreSQL のテストも）
`cargo clippy --all-targets`, `cargo fmt --check`	lint と整形（`max_width = 99`）
`cargo xtask check`	関数 70 行、ファイル 1,408 行、行 99/108 桁の上限。タブ禁止
`cargo xtask browser`	ヘッドレス Chromium での確認（`sibuna-browser`）
`cargo xtask bench <primitives|product|latency>`	リリースモードのベンチマーク。結果は `benchmarks/results/<command>-latest.json`
`cargo build -p sibuna-pow-wasm --target wasm32-unknown-unknown --profile wasm`	ブラウザー用モジュールのみ（コンソールのインターフェースには `-p sibuna-console-ui`）
`cargo run -p sibuna-geoip --bin geoip-snapshot -- ...`	国のデータを検証し、埋め込み可能な GeoIP のスナップショットを書き出す
`guidedog build` / `guidedog dsc`	本書 / SID の記録

デーモンのビルドスクリプト（`cmd/sibuna/build.rs`）は、二つの WebAssembly モジュールを `wasm` プロファイルでワークスペースのターゲットディレクトリにビルドし、コンソールのインターフェースが 768 KiB を超えるとビルドを失敗させます。`wasm32-unknown-unknown` ターゲットのないビルドでは、`SIBUNA_POW_WASM` と `SIBUNA_CONSOLE_WASM` で代わりにビルド済みのモジュールを指定します。`SIBUNA_GEOIP_DATA` は埋め込む GeoIP のスナップショットを指定します。これはビルド時にデコードされ、コンソールなしのビルドはこれを拒否します（`GEOIP001`）。

演習 10.1. 解答の途中で Wi-Fi からモバイル回線に切り替えた後、クライアントがタイトル `CLIENT FINGERPRINT MISMATCH` の `400` を受け取りました。トークンとチャレンジの形式から原因を説明し、穏当に回復するための中間ページへの最小の変更を提案してください。

不変条件を説明してください. 見返さずに、Sibuna の前に置くリバースプロキシがオリジンではなくデーモンにルーティングしなければならないエンドポイントを挙げ、それぞれが必要な理由を述べてください。

第 XI 部

早見表

ターミナルのそばに置いてください。ビルド、実行、設定、ルーティング、観測、診断を扱います。完全な表は第 X 部にあります。この早見表には、運用者が入力するものをまとめています。

ビルド

```
cargo build --release -p sibuna                          # デーモン+SQLite+コンソール
cargo build --release -p sibuna --no-default-features    # インメモリ、外部クレートなし
cargo build --release -p sibuna --features postgres-tls  # 共有PostgreSQLのクラスター
rustup target add wasm32-unknown-unknown   # またはSIBUNA_POW_WASM / SIBUNA_CONSOLE_WASM
cargo test            # 単体、稼働中のデーモン、ストレージ、コンソールのテスト
cargo clippy --all-targets && cargo fmt --check
cargo xtask check     # 70行 / 1,408行 / 99桁の検査
cargo xtask browser   # ヘッドレスChromiumでの検査
cargo xtask bench product   # wrkによる製品全体の測定 -> benchmarks/results/
guidedog build        # 本書（guidedog dsc：SIDの記録）
```

実行

```
# Shield（標準）：PoW + WAF、:3000へプロキシ
sibuna -p 8080 -u 3000 -s /etc/sibuna/secret
# Gate：PoWのみ
sibuna --gate -p 8080 -u 3000 -s /etc/sibuna/secret
# Nginx/Caddyの後ろでforward auth（X-Forwarded-Forを信頼する）
sibuna -m forward_auth --host 127.0.0.1 -p 8080 -s /etc/sibuna/secret
# Edge：<dir>/sibuna.dbに永続的なポリシー、レピュテーション、フォレンジック
sibuna -D /var/lib/sibuna -s /etc/sibuna/secret
# 信頼できるイングレスからのJA4（Cloudflareのヘッダー。nginxモジュール：x-ja4、ATS：ja4）
sibuna --ja4-header cf-ja4 --trust-forwarded -p 8080 -u 3000 -s /etc/sibuna/secret
# クラスターのメンバー：共有PostgreSQL、一意なノード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
```

コンソール（オプトインのプレビュー）

```
# デーモンを停止した状態で初期化し、その後コンソールを起動する
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
```

http://127.0.0.1:19446/console/を開き、一時パスワードを変更します。--console-location 1.3521,103.8198 を加えると、ノードを地球儀上に配置します。リモートからのアクセスには、HTTPS プロキシ、明示的なオリジン、信頼するプロキシの CIDR が必要です。第 IX 部に従ってください。SID 0007 は Committed です。コンソールはヘッドレス Chromium でのブラウザーのワークフローに合格しています（SID 0010、基準 5）。正式な性能分離の判定はまだ完了していません（第 VIII 部）。

まず押さえるべきフラグ

フラグ	既定値	意味
-d, --difficulty	16	作業ビット（PoSW の深さ = ビット - 3）
-a, --algorithm	posw	posw または hashcash
--posw-challenges	16	証明あたりの開示数
--token-ttl	86400	セッションの秒数
--challenge-ttl	300	チャレンジの秒数
--token-scheme	mac	mac または ed25519
--rate-limit / --rate-window	100 / 10	窓あたりの GCRA のバースト
--ban-seconds	3600	ハニーポットによる禁止
-w, --workers	CPU 数	受付スレッド
--max-connections	1024	超えると 503
--max-connections-per-ip	64	相手アドレスごと（IPv6 では--ipv6-prefix ごと）、超えると 503
--trusted-hops	1	クライアント = 右から n 番目の X-Forwarded-For の項目
--ja4-header	無効	イングレスの JA4 フィンガープリントを持つヘッダー。信頼する場合にだけ読む
--idle-timeout	15	回収の秒数
-P, --policy-file	なし	JSON のポリシー
--trust-forwarded	無効	forward auth では有効
--secure-cookie	無効	Secure を付加

/__sibuna/配下のエンドポイント

ルート	応答
challenge	中間ページの HTML
challenge.json?need=&path=	{id, algorithm, difficulty, challenges, expires_at}
verify（POST）	{challenge_id, nonce | proof} -> 200 + Set-Cookie、それ以外は 400
wasm/sibuna-pow.wasm	15,181 バイトのソルバー
worker.js	WASM と JS の証明器を持つワーカー
honeypot	呼び出し元を禁止して 403。サブリソースやクロスサイトの取得には 404
health	{"status":"ok",...}
metrics	Prometheus のテキスト

これらはすべてオリジンではなく、デーモンにルーティングしてください。

ステータスコード

コード	発生条件
200	許可（forward auth）、中間ページ、内部ルート
400	不正な形式のリクエストや対象、拒否された解答（診断のボディー）
401	チャレンジが必要：HTML 以外のクライアント、または forward auth
403	ポリシーまたは WAF による拒否、禁止、ハニーポット
404	不明な/__sibuna/のルート、拒否されたハニーポットへのアクセス
413	64 KB のバッファーを超える解答のボディー
414	8 KiB を超える、報告されたチャレンジの URL
417	サポートしないリクエストの expectation
429	GCRA の制限、Retry-After 秒
431	16 KB を超えるヘッダー部
501	chunked 以外の転送コーディング
502	オリジンに到達できない、または不正な形式
503	接続数の上限に到達、またはノードがドレイン中

ヘッダー

- 上流（プロキシ）：X-Forwarded-For（ソケットの相手。信頼する場合は受信した連鎖の後ろに追加）、X-Forwarded-Proto、X-Real-IP（判定したクライアント）、X-Sibuna-Status: PASS、X-Sibuna-Rule: <rule>。ホップごとのフィールドと別形式の転送フィールドは、アンダースコアの綴りも含めて取り除きます。
- *forward-auth* の応答：X-Sibuna-Status、X-Sibuna-Rule、X-Sibuna-Rule-Hash。
- チャレンジの応答：X-Sibuna-Status: CHALLENGE。
- *Cookie*：__sibuna_token=<75 chars, 139 with ed25519>; Path=/; Max-Age=<ttl>; HttpOnly; SameSite=Lax[; Secure]

ポリシーファイルの骨組み

```
{ "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：フィンガープリント全体、または接頭辞と*。フィンガープリントがなければ一致しない。パターン：*または.*は任意、^...$は完全一致、末尾の*は接頭辞、先頭の/はパスの完全一致、それ以外は大文字小文字を区別しない部分文字列です。パスは正規形で照合します。順序：WAF -> トライの deny/allow -> ルール -> スコア -> バイパス -> トライの challenge -> bots -> 既定。

SQL で見るストレージ（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）。動作名は小文字です。スコア≤ −50-> 拒否の接頭辞、≥ 50-> 許可。クラスター：全ノードで同じ秘密、一つの PostgreSQL データベース、--features postgres-tls、1 から 8,388,607 の一意な N を指定した--database-url-file <file> --cluster-node N。メンバーリスト、リーダー、クラスター内の TLS はありません（SID 0011）。

診断

タイトル	意味
MALFORMED CHALLENGE	ID が編集されたか切り詰められた。新しいものを取得する
INVALID CHALLENGE TAG	このシードまたはノードが発行したものではない
CHALLENGE EXPIRED	TTL より古い、または時計のずれが 60 秒超
CLIENT FINGERPRINT MISMATCH	アドレスまたは User-Agent が変わった
DIFFICULTY NOT MET	nonce のゼロビットが足りない
INVALID PROOF OF SEQUENTIAL WORK	開示が根に到達しない
WRONG SOLUTION TYPE	PoSW に nonce、または Hashcash に証明
DOUBLE SPEND PREVENTED	チャレンジはすでに使用済み
STORE FULL	使用済み集合が飽和。TTL を下げるか--spent-tags を上げる
NETWORK BINDING MISMATCH	Cookie が別のクライアントにコピーされた
INVALID COMMAND LINE	不明なオプション、不正な値、または削除された Zaxonlite のオプション

アラートを設定すべきメトリクス：denied、 rate_limited、 overloaded、 upstream_errors、 incidents_dropped、 incident_write_failures、 bans_refused、 forwarded_invalid（すべて sibuna_*_total）。

覚えておく数値

チャレンジ ID は 70 文字（36 バイトのペイロード + 16 バイトのタグ）。トークンは 75 文字（40 + 16）、Ed25519 では 139 文字。証明のバイト数は$32(1 + t(n + 1))$。PoSW の深さ = ビット - 3、範囲は 4 から 24。リクエストバッファーは 64 KB。ヘッダー部の上限は 16 KB。検査するボディーは 8 KB。使用済み集合は 1,048,576 タグ（--spent-tags）。レートのセルは表あたり 262,144（--rate-table-cells）。禁止は 65,536 スロット（--ban-table-slots）。クライアントアドレスあたり 64 接続。インシデントリングは 512。バッチは 32。キャンペーンの距離は最新 16,384 個の埋め込みベクトルに対して 0.35。keep-alive は接続あたり 4,096 リクエスト。オリジンのプールは 256 ソケット。ストレージのジョブキューは 32。

イングレスのルーティング

第 IX 部「イングレスの背後での forward auth」の Nginx または Caddy のレシピを使ってください。/__sibuna/*は直接ルーティングします。アプリケーションのリクエストは、元の URI、メソッド、信頼できるクライアントアドレスで認可します。受信した Sibuna の監査用ヘッダー

は、認可の結果で置き換えます。Nginx では、チャレンジ、429、認可が利用できない場合を明示的に扱う必要があります。

`reverse_proxy` は、許可した HTTP/1.1 のボディーと WebSocket を運びます。`forward_auth` は許可を与え、それらはイングレスが運びます。認可で省かれたボディーとオリジンの応答は、Sibuna が検査することも数えることもできません。TLS はイングレスで終端します。ネイティブの HTTP/2 は先送りしています。アップロードは Content-Length か chunked コーディングを使います。検査が見るのは最大 8 KiB の先頭部分で、ファイルのバイトは中身を見ません。

解答例（抜粋）

これらの解答は議論を確かめるためのものであり、演習の代わりではありません。結果が異なるときは、まず前提を比べます。時間区間の端点、何を作業として数えるか、書き込みがコミットされたかどうかは、食い違いのよくある原因です。

第 I 部：コストと確率

1.1.100 万回の試行を 100 件のリクエストで割ると、リクエストあたり 10,000 回の試行です。総リクエスト数が 100 のままなら、セッションを共有してもこの計算は変わりません。実際のシステムでは、二つ目のプロセスがフィンガープリントとルーティングの結び付きを満たせるかどうかを判断しなければなりません。その条件は償却の式の外にあります。
1.2.超過分は$1000 - 600 = 400$レコード/s です。512 レコードの空のキューは、流体モデルでは$512/400 = 1.28$秒を稼ぎます。実際の生産者はバーストで到着し、消費者はバッチでコミットするので、最初に破棄が起きる正確な時刻は到着とコミットの時刻に依存します。

第 II 部：先行事例

2.1.一つのソースを二回コンパイルしたものは、ステートメント、nonce のエンコーディング、木の形、開示の配置について自分自身と食い違うことはありません。食い違うにはコンパイラーのバグが必要です。ただし、構成が健全かどうか（暗号のレビュー）、ブラウザーのランタイムがそれを正しく実行するかどうか（WebAssembly と JavaScript の証明器をバイト単位で比較することで確かめます）、パラメーターが較正されているかどうか（測定）については、何も教えてくれません。
2.2.(a) Sibuna Gate または Anubis で、バイナリ一つ。(b) Sibuna Shield、SafeLine、Cloudflare（Pro プラン以上）。(c) 共有 PostgreSQL データベース（`--database-url`）を使う Sibuna Edge、共有 Valkey ストアを使う Anubis（状態のためであり、禁止のためではありません）、または Cloudflare。Docker ホストもサブスクリプションも使わずに三つすべてを満たす単一の製品は、ノードが一つの PostgreSQL データベースを共有する Sibuna Shield（`--database-url` と `--cluster-node`）だけです。決め手になるのは「実行形態」「検査」「複数ノード」の行です。

第 III 部：暗号

3.0.裾の見積もりは$e^{-4.6} \approx 0.010$です。nonce 空間を分割すると、試行が完了する速さは変わりますが、独立な各試行の成功確率は変わりません。重複した試行や調整のための作業によって、実際の並列実装の効率は下がることがあります。
3.1.各コミットメントが確率qで成功するなら、g個すべてが失敗する確率は$(1-q)^g$です。少なくとも一つが成功する確率は$1-(1-q)^g$です。この式は、コミットメントの構築、クエリの予算、試行間の依存、特定の受理確率を持つコミットメントを見つけるのに必要な作業を数えていません。

3.2.深さ 17 で開示が 16 回なら、証明は$32(1 + 16 \times 18) = 9248$バイトです。開示が 32 回なら$32(1 + 32 \times 18) = 18464$バイトです。開示のバイト数は 2 倍になりますが、32 バイトの根は相変わらず一回だけ送られます。

3.3.作業レベルがなければ、安いルートで得たセッションで、より多くの作業を要求する高価なルートへのリクエストが許可されてしまいます。`timestamp` がなければ、トークンを「未来の時刻で」発行して、ポリシーより長く生き延びさせることができます。`expiry` がなければ、トークンは決して失効しません。`rule_hash` がなければ、上流はどのルールがクライアントを許可したかを知れません。`fingerprint` がなければ、別のマシンにコピーした Cookie が受理されてしまいます。タグが五つすべてを覆わなければならないのは、タグの外に残したフィールドは自由に書き換えられ、検証者にはそれが分からないからです。

第 IV 部：プロトコル

4.1.二つのブラウザーはアドレスと User-Agent を共有するので、フィンガープリント、レート制限のセル、禁止のスロットを共有します。チャレンジ（それぞれが自分の nonce を持つ自分のレコードを取得します）、使用済み集合の項目、Cookie は共有しません。もう一方のブラウザーの Cookie を再利用するには、NAT の背後の攻撃者はそれをもう一方のマシンから読み取る必要があります。そうすればフィンガープリントは一致してしまいます。だからこそ、共有アドレスでは結び付きよりも `HttpOnly` と `SameSite` が重要になります。

第 V 部：状態と順序

5.1.ヘッダー部とボディーの最初の部分（64 KB のうちヘッダー部の後に収まる分まで）は接続バッファーにあり、`write_all(request.body)`で送られます。残りのバイトがまとめてバッファーに入ることはありません。`relay_body` がクライアント側のリーダーから 16 KB ずつ読み、それぞれをオリジンに書き込みます。ヘッダー部を先に送らなければならないのは、オリジンがボディーの前に `Content-Length` を必要とするからであり、また監査用のヘッダーがヘッダー部に含まれるからです。

5.2.送出間隔は 100 ms、バースト許容量は 9,900 ms です。アイドル状態のクライアントは 100 件のリクエストをすぐに送れ、その後は 100 ms ごとに 1 件です。端点を含む 3,000 ms までの上限は 130 です。10 秒の固定カウンターなら、最初の窓で 100 件、窓が切り替わった瞬間にさらに 100 件を許します。スライディングログなら任意の 10 秒でちょうど 100 件を許しますが、クライアントごとに 100 個のタイムスタンプを保存しなければなりません。

第 VI 部：アルゴリズム

6.1.名前に含まれるアポストロフィーや、`order` という名前の普通のパラメーターは、SQL の式であることを示しません。部分文字列による検出器は、どちらも構文と取り違えかねません。構造による検出器は、演算子やリテラル同士の関係など、追加の証拠を探します。それでも上限のある検査であり、あらゆるアプリケーション言語のパーサーではありません。

6.2.LDAP のフィルターは、括弧、`|`、`&`、`!`、`=`から作られます。検出器には`(`と`)`のためのバイトクラス（新規）が必要で、`|`と `&` には既存の`=`とシェルの区切り文字のクラスを再利用できま

す。SQL 検出器が引用符か恒真式を要求するのと同じ理由で、括弧一つではなく、)(や(|(のような並びに反応すべきです。

第 VII 部：ブラウザーエンジン

7.1.既定の 16 作業ビット（深さ 13、開示 16 回）では、WebAssembly で 20 ms と仮定した時間が、ノート PC の JavaScript では$50 \times 20 = 1000$ms、つまり 1 秒になり、遅いスマートフォンでは 4 秒になります。これらの数値は仮定です。ポリシーを設定する前に、自分たちのクライアントの端末で測定します。妥当なポリシーは次のとおりです。ページに `fallback` メッセージを転送させ、それをサーバー側でメトリクスとして記録し、全員ではなく、WebAssembly を遮断することが分かっているクライアントに一致するルールについてだけ難易度を下げます。

第 VIII 部：評価

8.2. Gate の `admitted` の行では$11.84 \times 211,345 / 10^{6} \approx 2.50$で、稼働中のコア数は四つのうち 2.50 です。したがってデーモンは CPU 律速ではなく、上限を決めたのは負荷生成器です。リクエストあたり 11.84 us の CPU が 1 マイクロ秒未満の分類コストをはるかに上回るのは、製品の行には、カーネルによるソケットの読み書き、パーサー、セッションタグの確認、メトリクスの加算、応答の整形、ライターのフラッシュ、負荷生成器と接続スレッドの間のスケジューラーによる受け渡しも含まれるからです。プリミティブのスイートはそのどれも計時しません。

第 IX 部：運用

9.1.受領記録の別の更新より先にデータがコミットされると、その間にクラッシュしたときに古い受領記録が残り、再試行で効果がもう一度適用され得ます。受領記録が先にコミットされると、データの書き込みの前にクラッシュしたとき、再試行は存在しなかったデータを飛ばしてしまい得ます。一つのアトミックなトランザクションで、両方の隙間がなくなります。トランザクションの確認応答が失われた場合には、保持した再試行が引き続き必要です。
9.2.`ip_rules: {"10.20.0.0/16": "ALLOW"}`（またはスコア 100 の `ip_reputation` 行）でステージング環境を許可します。ルール`{"name": "admin", "path": "/admin/*", "action": "CHALLENGE", "challenge": {"difficulty": 20, "algorithm": "posw"}}`で作業を要求します。`headers: {"X-Partner-Key": ".*"}`に一致し、`ALLOW` と`"limits": {"rate": 30, "window_seconds": 10}`を持つルールで、パートナーを 10 秒あたり 30 リクエストで許可します。ルールのクォータは、そのルールに一致するすべてのリクエストについて、クライアントごとのデーモンの制限（`--rate-limit`、`--rate-window`）の後に確認します。したがって、デーモンの制限は少なくともそれと同じだけ緩くなければなりません。

第 X 部：リファレンス

10.1.チャレンジレコードは、それを取得したアドレスの鍵付きフィンガープリントを運びます。解答は新しいアドレスから届くので、`verify_and_mint` は証明を見る前に `FingerprintMismatch` を返します。最小限の回復策は中間ページにあります。タイトルが `CLIENT FINGERPRINT MISMATCH` の

400 を受け取ったら、エラーを表示する代わりに新しいチャレンジを取得して解き直します。すでに行った作業は移せないからです。

さらに調べるために

使い捨てのテストで、不変条件を一度に一つずつ変えてみてください。スナップショットの再確認を取り除く、インシデントの受領記録を省く、レートの間隔を切り捨てる、といった変更です。テストを実行する前に、失敗の起き方を予測します。有用な回帰テストは、正常系の出力を確認するだけではありません。省いた条件がなぜ必要なのかを実証します。

用語集

許可
: リクエストをオリジンへ転送する判断です。Sibuna では、有効なセッション、`ALLOW` ルール、バイパス対象の静的パス、レピュテーションの `allow` 接頭辞のいずれかによって許可します。許可が WAF の拒否を覆すことはありません。

Aho-Corasick オートマトン
: 入力を一度走査するだけで、固定したパターン集合のどのパターンでも見つける有限オートマトンです。Sibuna は失敗リンクを密な遷移表に組み込むので、各バイトのコストは表の読み込み一回です。

禁止テーブル
: 禁止したクライアントのアドレスを期限とともに持つ、4,096 個のロックフリーなスロットです。ハニーポットと、クラスターで共有するレピュテーションが書き込み、すべてのリクエストで読みます。

キャンペーン
: ペイロードの埋め込みベクトルが互いにコサイン距離 0.35 以内にあるセキュリティインシデントのまとまりです。ヒューリスティックなグループ分けであり、同じ発信元であることの証明ではありません。

チャレンジレコード
: クライアントが解くべき、70 文字の状態を持たない識別子です。バージョン、アルゴリズム、難易度、開示数、発行時刻、クライアントのフィンガープリント、PRF nonce、ルールのハッシュ、16 バイトの鍵付き BLAKE3 タグを含みます。発行してもサーバーには何も書き込みません。

難易度、作業ビット
: 二つの層の規模を決める一つの整数です。Hashcash では先頭のゼロビットの数、逐次作業では木の深さに 3 を足した値です。

Edge
: `--data-dir`（一つのノード上の SQLite ファイル）、または`--database-url` と`--cluster-node`（クラスターで共有する一つの PostgreSQL データベース）を使う Sibuna のデプロイです。どちらの運用構成にも、永続的なポリシー、共有レピュテーション、インシデントのフォレンジックを提供します。

エンジンスロット
: 公開中のポリシーエンジンを参照カウント付きポインターとして置く場所です。ワーカーは分類の間、参照を保持します。ストレージスレッドはリクエストの経路の外で新しいエンジンを構築し、ポインター交換で公開します。古いエンジンは、最後の読み手が参照を手放したときに解放されます。

フィンガープリント
: クライアントのアドレスと User-Agent の鍵付きハッシュです。チャレンジとトークンに結び付けるので、どちらも別のクライアントから再利用できません。

Forward auth
: イングレスが Sibuna に、リクエストを許可するか（`200`）、チャレンジ、拒否、制限するか

（401、403、429）を問い合わせ、オリジンへのプロキシはイングレス自身が行うデプロイ方式です。

Gate
作業証明による許可、セッション、ルール、レピュテーション、GCRA による制限、禁止を備え、アプリケーションの検査を持たない運用構成です（--gate）。

GCRA
Generic Cell Rate Algorithm です。クライアントごとに一つの理論到着時刻を持ち、N個のバーストを許可した後は送出間隔ごとに一つのリクエストを許可します。窓の境界はありません。

Hashcash
第一層です。チャレンジ、コロン、十進数の nonce を連結したものの SHA-256 が先頭にb個のゼロビットを持つような nonce を探します。期待試行回数は2^b回で、検証はハッシュ一回です。

ハニーポット
/__sibuna/honeypot を指す見えないリンクです。たどったクライアントは禁止され、レピュテーション−100のインシデントが記録されます。

アイドル接続回収スレッド
--idle-timeout より長く無通信の接続を閉じるスレッドです。これにより、遅いクライアントが接続スレッドを占有し続けることはできません。

中間ページ
保護されたページの代わりに配信する組み込みの HTML ページです。チャレンジを取得し、Web Worker で解き、解答を送信して再読み込みします。

JA4
FoxIO が定義した 36 文字のフィンガープリントで、クライアントが TLS 接続をどのように開くか（プロトコル、TLS バージョン、サーバー名、暗号スイートと拡張の数、ALPN、ソートした暗号スイートと拡張のハッシュ）を表します。Sibuna はこれを計算しません。信頼するイングレスが--ja4-header で指定したヘッダーで転送する値を読み、ルールは ja4 フィールドでこれと照合します（SID 0014）。チャレンジとトークンに結び付けるクライアントのフィンガープリントとは無関係です。

鍵付き BLAKE3
Sibuna の鍵スケジュール、チャレンジタグ、セッションタグ、フィンガープリント、チャレンジの nonce を支える擬似ランダム関数です。

MAC トークン
標準のセッション Cookie です。40 バイトのペイロード（バージョン、作業レベル、タイムスタンプ、期限、ルールのハッシュ、フィンガープリント）と 16 バイトの鍵付き BLAKE3 タグからなり、定数時間で検証します。

作業レベル
セッションの保有者が実際に解いた方式と作業ビットで、トークンに記録されます。ルートがセッションを許可するのは、そのレベルがルートの要件に達するときだけです。

開示
逐次作業証明における、一つの葉のラベルと、その葉から根までの経路上の兄弟ラベルの組です。検証者は経路を再計算し、コミットメントと比較します。

逐次作業証明（PoSW）
: 第二層です。各ラベルがそれ以前のすべてのラベルに依存する Cohen-Pietrzak のハッシュグラフを使い、証明者のコア数にかかわらず$2^{n+1}-1$回の逐次ハッシュを強制します。検証は$t(n+1)$回のハッシュです。

レピュテーショントライ
: ポリシーファイルと `ip_reputation` テーブルから得た `allow`、`deny`、`challenge` の接頭辞を保持する 128 ビットの基数トライです。

Robin Hood 使用済み集合
: 解かれたチャレンジのタグを持つ、容量固定のオープンアドレス法のテーブルです。有効な解答が二度提出されることを防ぎます。

ルールのハッシュ
: チャレンジを要求したルールの名前の 64 ビットハッシュです。チャレンジ、トークン、`X-Sibuna-Rule-Hash` ヘッダーで運ばれます。

Shield
: 標準の運用構成です。Gate に意味解析ファイアウォールを加えたものです（`--shield`）。

SID
: Sibuna Discussion の記録です。`docs/sid/records` にある論文形式の設計文書で、システムの一部分を支える前提、証明、測定を示します。

運用構成
: Gate と Shield という二つの製品の形のいずれかです。ストレージとクラスタリングはどちらでも選択できます。

WEIGH
: 判断を下す代わりに、リクエストのスコアに符号付きの重みを加えるルール動作です。合計がしきい値以上になると、追加の作業ビット付きでチャレンジするか、拒否します。

Zaxonlite
: SID 0005 の組み込み SQLite 複製エンジン（Multi-Paxos）です。SID 0011 で SQLite と PostgreSQL によるストレージに置き換えられました。そのクラスターオプションを指定すると、現在は起動が `RemovedOption` で停止します。

参考文献

- 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." 技術報告, 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*. 博士論文, 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、および *HTTP/1.1*, RFC 9112. IETF, 2022.
- FoxIO. *JA4+ Network Fingerprinting*. https://github.com/FoxIO-LLC/ja4, 2023（JA4 TLS クライアントフィンガープリント）.
- 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、および"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*. 仕様, 2020.
- Vyukov, D. "Bounded MPMC queue." 1024cores.net, 2011（インシデントキューで使う、シーケンス番号付きのリング）.
- 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 0001 から 0014、`docs/sid/records`、2026。プロセス、基礎アーキテクチャ、宣言的ポリシーエンジン、意味解析による検査、Zaxonlite ストレージのアーキテクチャ、数学的基礎、コンソール管理インターフェース、AI ボットのトラフィック監視、チャンク形式のリクエストボディー、Rust への移植、sqlx によるストレージ、rustls によるコンソールの外向きの TLS、コンソールのストレージ、イングレスから受け取る信頼できる JA4 フィンガープリントを扱います。

索引

A
Aho-Corasick オートマトン, **149**

E
Edge, **149**

F
Forward auth, **149**

G
Gate, **150**
GCRA, **150**

H
Hashcash, **150**

J
JA4, **150**

M
MAC トークン, **150**

R
Robin Hood 使用済み集合, **151**

S
Shield, **151**
SID, **151**

W
WEIGH, **151**

Z
Zaxonlite, **151**

ア
アイドル接続回収スレッド, **150**

エ
エンジンスロット, **149**

キ
キャンペーン, **149**

チ
チャレンジレコード, **149**

ハ
ハニーポット, **150**

フ
フィンガープリント, **149**

ル
ルールのハッシュ, **151**

レ
レピュテーショントライ, **151**

中
中間ページ, **150**

作
作業レベル, **150**

禁
禁止テーブル, **149**

許
許可, **149**

逐
逐次作業証明（PoSW）, **151**

運
運用構成, **151**

鍵
鍵付き BLAKE3, **150**

開
開示, **150**

難
難易度、作業ビット, **149**