GitHub Actionsで実現する DevSecOps入門

WHITE PAPER



Start Challenge Consulting co., ltd.

挑む力が、世界を照らす

発行：2026年8月

Strictly Confidential

リリース直前のセキュリティ確認は、手戻りを大きくする

セキュリティ確認を工程の最後に置く限り、修正はリリース直前に集中し、開発速度と品質の双方を圧迫する。
確認をCI/CDへ組み込み、コミットとプルリクエストの単位で自動実行することが解決の起点となる。

現状

・機能開発は継続的に進む一方、セキュリティ確認は特定の工程や担当者に集中しやすい

・確認の実施時期・観点・判定基準が明文化されず、レビュー品質が担当者に依存する

重要課題

・設計や依存関係に起因する問題が終盤で判明し、修正範囲が広がる

・OSSライブラリや外部Actionを経由した脆弱性・改ざんの持ち込みを把握できない

・認証情報がコードや実行ログへ残る

推奨施策

・GitHub Actionsのワークフローとして、検査をリポジトリ内にコード化する

・プルリクエスト単位で自動実行し、結果を必須ステータスチェックとしてマージ判定へ反映する

期待効果

・問題が小さい単位で早期に判明し、修正の影響範囲が限定される

・確認手順が設定ファイルとして残り、属人化を回避できる

・判定基準が全リポジトリで揃う

本資料で理解できること

① DevSecOpsの考え方と、GitHub Actionsが適している理由　② コミットから運用までの検査ポイントの全体像
③ 各検査（SAST・依存関係・シークレット・コンテナ・IaC）の設定方針　④ マージ判定・例外管理・段階導入の運用設計

事実：GitHub公式ドキュメントは、code scanningを「リポジトリへの統合前に修正できるようにする」機能と位置づけている（出典：GitHub Docs / About code scanning）。
解釈：ゆえに確認工程の前倒しが有効である。

セキュリティを専門部門の工程から、開発ライフサイクル全体の常時活動へ

DevSecOpsは、開発・セキュリティ・運用を統合し、問題の検出を可能な限り開発工程の早い段階へ移す考え方である。
差は「誰がいつ確認するか」ではなく、「確認が自動化され、判定基準がコードとして共有されているか」に現れる。

観点	従来型開発	DevOps	DevSecOps
確認の時期	リリース前の検査工程に集約	機能テストは継続的、セキュリティは終盤に残る	**コミット・プルリクエスト単位で継続的に実施**
実施主体	セキュリティ部門・外部診断	開発と運用が共同、セキュリティは別組織	**開発・運用・セキュリティが基準を共有**
確認手段	手作業のレビュー・チェックシート	CI/CDによるビルド・テスト自動化	**検査をワークフローとしてコード化し自動実行**
是正の起点	検査報告書の受領後	テスト失敗の通知後	**開発者の作業画面（PR）へ結果を直接返す**
判定基準	担当者の経験に依存しやすい	ビルド成否が基準、セキュリティ基準は別管理	**必須ステータスチェックとして明文化・共有**

OWASPが示すDevSecOpsの目的

「開発プロセスにシフトレフトのセキュリティ文化を促進すること」を目的とし、設計または実装上の脆弱性を可能な限り高速に検出することを理想目標とする。
パイプラインの初期ステップとして、クレデンシャル漏えい検知・SAST・SCA・IaCスキャン等を挙げている。

出典：OWASP DevSecOps Guideline。事実：目的および初期ステップの記述は同ガイドラインに基づく。
解釈：上表の3類型の対比は本資料による整理であり、単一の公式定義に基づくものではない。

コード・レビュー・実行基盤が同一プラットフォーム上にあることが利点となる

GitHub Actionsは、検査の実行条件をリポジトリ内のYAMLとして管理し、結果をプルリクエストの必須チェックへ直結できる。
これにより「検査の実施」と「マージの可否判断」が同一の統制下に置かれ、確認漏れと属人化を構造的に抑止できる。

① 一元管理

ソースコード、レビュー、CI/CDの実行が同一リポジトリ内で完結する。

ワークフローは**.github/workflows/** 配下のYAMLとして保存され、コードと同じ履歴・レビュー手順で管理できる。

② PR単位の実行

pull_request イベントは、PRのopen・synchronize・reopenを既定の契機として起動する。

変更差分が小さい段階で検査でき、修正範囲を限定できる。

③ ツール連携

code scanningは **SARIF**（静的解析結果の交換形式）というオープン標準に対応する。

第三者製ツールの検出結果も、同じ画面・同じ運用手順に取り込める。

④ 結果の返却

検査結果はチェック結果としてPR画面に表示される。

必須ステータスチェックに登録した場合、対象チェックが成功・スキップ・中立のいずれかでなければマージできない。

⑤ 設定のコード化

検査対象・実行契機・権限をYAMLで宣言する。

設定変更もPRレビューの対象となり、「誰がいつ検査条件を変えたか」を追跡できる。

出典：GitHub Docs「Workflow syntax」「Events that trigger workflows」「About protected branches（Require status checks before merging）」「About code scanning」。
事実：各機能の仕様は上記公式ドキュメントの記述に基づく／解釈：①〜⑤の利点整理は本資料による。

検査対象と実行契機を工程ごとに割り当て、重複と抜けを排除する

検査は工程ごとに対象が異なるため、同一の検査を全工程で回す必要はない。コミット時＝シークレット、PR時＝静的解析と依存関係、ビルド時＝コンテナイメージ、デプロイ前＝IaCと設定値、運用時＝アラート監視。この割り当てが実行時間と検出範囲を両立させる。

工程	① コミット	② プルリクエスト	③ ビルド	④ デプロイ前	⑤ 運用
実行契機 （イベント）	push （プッシュ時にブロック）	pull_request ＋ schedule（定期）	push（既定ブランチ） ／リリース作成	Environmentsの 承認ゲート通過時	アドバイザリ公開時 ／依存関係グラフ変化時
主な検査 （GitHub機能等）	シークレット検知 secret scanning / push protection	静的解析・依存関係検査 code scanning（CodeQL）/ dependency review / Dependabot alerts	コンテナイメージ検査 Trivy等でOSパッケージ・依存関係の既知脆弱性を検出し、SARIFで取り込む	IaC・設定値検査 Trivyのmisconfigurationスキャナ等で公開範囲・暗号化・権限を確認	アラート監視と継続改善 Dependabot alerts / security updates による自動修正PR
不合格時の取り扱い	**プッシュを拒否** （理由付きの回避は記録・通知される）	**必須チェック失敗** ＝マージ不可	**配信対象から除外** （重大度により判断）	**デプロイを停止** （承認者の判断を要する）	**対応期限を設定し** 検出数・対応時間を測定

出典：GitHub Docs「Events that trigger workflows」「Push protection」「Dependency review」「About Dependabot security updates」、Trivy公式リポジトリ（aquasecurity/trivy）。
事実：各機能の仕様は上記に基づく／提案：工程別の割り当ておよび不合格時の取り扱いは本資料による設計案。

検査導入の前に、検査結果を強制できる状態をリポジトリ側で作る

ブランチ保護と権限の最小化がなければ、検査を追加してもレビューを経ずに変更が入り、判定を回避できてしまう。
まず「マージ経路の一本化」と「権限の最小化」を確立し、その経路上に検査を配置することが導入順序として合理的である。

整備項目	設定内容	留意点（公式仕様）
ブランチ保護ルール	既定ブランチへの直接pushを禁止し、変更はPR経由に限定する	rulesetsとbranch protection ruleは併存し、該当する全ルールが適用される。同一ブランチに適用できるbranch protection ruleは1つ、rulesetは複数を重ねられる
レビューの必須化	マージ前に指定人数の承認を必須とする	「Require review from Code Owners」を有効化した場合、対象ファイルのオーナーのうち1名の承認で要件を満たす（全員の承認は不要）
ステータスチェックの必須化	検査ワークフローを必須チェックに登録する	対象チェックが successful・skipped・neutral のいずれかでなければマージ不可。ベースブランチとの同期を強制するStrictモードの有無を選択できる
CODEOWNERSの整備	パス単位でレビュー担当を明示。**.github/workflows** も対象に含める	GitHubは .github ／リポジトリルート／docs の順にファイルを探索する。指定できるのはwrite権限を持つユーザー・チーム等。ファイルサイズ上限は3MB
権限の最小化	Read／Triage／Write／Maintain／Adminから最小の役割を割り当てる	Readはコードを書かない参加者、Writeは実際にpushする者、Maintainは破壊的操作を除く管理、Adminは削除・セキュリティ設定を含む完全アクセスと定義される
強制pushの制限	履歴の書き換えとブランチ削除を禁止する	保護ブランチでは既定でforce pushと削除がブロックされる。設定により許可することも可能なため、許可状態になっていないかを確認する

出典：GitHub Docs「About protected branches」「About rulesets」「About code owners」「Repository roles for an organization」。
事実：上表の留意点は各公式ページの記述に基づく／提案：整備の順序付けは本資料による。

コードを実行せずに問題を検出し、統合前の修正へつなげる

SASTはソースコードを実行せずに解析し、脆弱性を検出する手法である。GitHubのcode scanning（解析エンジンCodeQL）は「統合前に修正できるようにする」機能と位置づけられており、PR時の差分検査と定期的な全体検査の併用が設計の要となる。

設定方式の選択（Default setup と Advanced setup）

Default setup：解析対象言語・クエリスイート・スキャン契機をGitHubが自動選択する。導入初期に適する。

Advanced setup：github/codeql-action を用いたYAMLを生成し、対象範囲・実行契機・除外設定をカスタマイズする。外部CIや第三者ツールのSARIF取り込みも可能。

CodeQLの対応言語（公式サポート）

C/C++、C#、Go、Java/Kotlin、JavaScript/TypeScript、Python、Ruby、Rust、Swift、および **GitHub Actions のワークフロー**。

ワークフロー自体が解析対象に含まれるため、パイプラインの設定不備もSASTの検出範囲として扱える。

実行契機の使い分け

pull_request：変更差分に対する検査。開発者が修正しやすい単位で結果を返すことを目的とする。

schedule：cron式による定期実行。新しい検出ルールの追加や、長期間変更されていないコードへの再評価を担う。両者を併用するワークフローが公式に提示されている。

マージ判定への反映と誤検知への対応

・検査を**必須ステータスチェック**として登録することでマージをブロックできる（branch protection／rulesetsのいずれでも登録可）
・修正が反映されるとアラートは自動的にクローズされる
・誤検知は「担当者・理由・再評価予定日」を記録して閉じる運用ルールを定める（対応期限の設計はP12参照）

出典：GitHub Docs「About code scanning」「About code scanning with CodeQL」、CodeQL Docs「Supported languages and frameworks」。提供条件：code scanningはパブリックリポジトリでは既定で無償、Organization所有のプライベート／内部リポジトリではGitHub Code Securityが必要。提案：誤検知の記録項目は本資料による運用案であり、公式の必須要件ではない。

脆弱性は自社コードだけでなく、依存関係とActions自体からも持ち込まれる

脆弱性は「既に取り込んだもの（アラート）」と「これから取り込むもの（PR差分）」の双方を押さえる必要がある。さらにワークフローが呼び出す外部Action自体も改ざんの対象となるため、コミットSHAによるバージョン固定が要件となる。

機能	役割	運用上の要点（公式仕様）
Dependabot alerts	既に取り込んだ依存関係の脆弱性を把握する	GitHub Advisory Databaseの新規アドバイザリと依存関係が一致した場合、または依存関係グラフが変化した場合にアラートを生成。SecurityタブとDependabotタブに表示される
Dependabot security updates	脆弱な依存関係を修正版へ上げる自動PRを作成する	アラートを起点に最小修正バージョンへの更新PRを生成。GitHub Actions自体も対象となる。dependabot.ymlで挙動を調整でき、グループ化にも対応
Dependabot version updates	脆弱性の有無に関わらず、定期的に最新版へ追随する	スケジュールに従いsemverに基づく更新PRを生成。既定で3日間のクールダウン期間があり、これはsecurity updatesには適用されない
Dependency review	マージ前に依存関係の差分を検査する	PRでの追加・削除・更新を表示し、既知脆弱性があれば既定でチェックを失敗させる。必須チェックに登録すればマージをブロックできる。重大度のしきい値やライセンスの許可・拒否リストも設定可能
外部Actionの固定	呼び出すActionの改ざんリスクを抑える	公式は「完全長のコミットSHAへのピン留めが、Actionを不変のリリースとして使う現時点で唯一の方法」と明示。Organization／リポジトリ単位でピン留めを強制するポリシーも設定できる

実例：外部Actionの侵害（tj-actions/changed-files、2025年3月）

botアカウントに紐づくアクセストークンが侵害され、悪意あるコミットがActionのタグに再付与された。23,000以上のリポジトリが影響を受け、CI/CDのシークレットがビルドログへ出力される事象が発生した。GitHubは2025年3月15日14:00 UTCに当該Actionを利用不可とし、同日22:00 UTCに全バージョンから悪性コードを除去して復旧した（CVE-2025-30066）。

出典：GitHub Docs「Supply chain security」「About Dependabot security updates」「About Dependabot version updates」「Dependency review」「Security hardening for GitHub Actions」、StepSecurity調査報告、GitHub Advisory GHSA-mrrh-fwg8-r2c3（CVE-2025-30066）。事実：いずれも上記出典の記述に基づく。

「入る前に止める」「入ったら検知する」「漏れたら失効させる」を揃える

認証情報は一度コミットされると全履歴に残るため、検知だけでは不十分である。プッシュ時のブロック（予防）、履歴全体の走査（検知）、
失効・再発行の手順（是正）を三層で設計し、加えて実行ログへの出力経路を閉じることが必要となる。

① 予防：push protection

CLIからのpush、GitHub UI上のコミット、ファイルアップロード、REST API等の操作を、シークレット検出時にブロックする。

既定ではwrite権限を持つ利用者が理由を付して回避でき、その際はアラート・監査ログ・所有者への通知が生成される。回避権限や審査者を細かく制御する **delegated bypass** も提供される。

② 検知：secret scanning

リポジトリの全ブランチ・全履歴を走査し、ハードコードされた認証情報を検出する。

対象はコードのみでなく、Issue／PRのコメント、Discussions、Wiki、シークレットgistを含む。新しい検出パターンが追加された際にも再走査される。パートナー対応のシークレットは、GitHubが発行元へ直接通知する。

③ 是正：失効・再発行・調査

公式は、シークレットがログに出力された、または漏えいの懸念がある場合、**当該ログを削除しシークレットをローテーションする**ことを明示している。

あわせて「定期的なローテーションと、不要になったシークレットの削除」を推奨。履歴に残った値は失効させる以外に無効化できない点を前提に手順を定める。

保管場所の使い分け（GitHub Secrets）

・シークレットは**リポジトリ／Environment／Organization**の3階層で作成できる。Organizationレベルはアクセス可能なリポジトリをポリシーで限定でき、GitHub Freeのプライベートリポジトリでは提供されない
・Environment secretsは、必須レビュー等の環境保護ルールと組み合わせて本番資格情報を分離する
・フォークからのPRでは **GITHUB_TOKEN** 以外のシークレットはランナーへ渡されない

ログへの認証情報出力を防ぐ

・登録済みシークレットの値は、実行ログ上で自動的に伏字化される
・ただし伏字化は文字列の完全一致で行われるため、**JSON等の構造化データをまとめてシークレット化すると伏字化に失敗する**。個別のシークレットとして登録する
・シークレット以外の値は **::add-mask::** コマンドで伏字化できる

出典：GitHub Docs「About secret scanning」「Push protection」「Using secrets in GitHub Actions」「Security hardening for GitHub Actions」。
提供条件：secret scanningはパブリックでは既定で無償、Organization所有のプライベート／内部リポジトリではGitHub Secret Protectionが必要。push protectionは個人向けとリポジトリ向けに分かれる。

検査対象をコードから「実行環境の定義」へ広げ、デプロイ前に止める

コードが安全でも、ベースイメージの既知脆弱性や、IaCの公開範囲・暗号化・権限設定の誤りは残る。コンテナイメージとIaCを検査対象に加え、結果をSARIFでcode scanningに集約すれば、単一の運用手順で扱える。

検査対象	確認する内容	実装上の要点
コンテナイメージ	イメージに含まれるOSパッケージおよびアプリケーション依存関係の既知脆弱性（CVE）	Trivyはスキャナとして「OSパッケージとソフトウェア依存関係（SBOM）」「既知脆弱性（CVE）」等を、対象としてコンテナイメージ・ファイルシステム・Gitリポジトリ・Kubernetes等を公式に挙げている
ベースイメージの管理	利用するベースイメージのバージョンが特定・追跡可能であること	可変タグではなくダイジェスト固定により、ビルドの再現性と改ざん検知性を確保する。※本項目はDocker公式ドキュメントによる直接の記述を確認できておらず、本資料の提案として扱う
Dockerfileの設定	危険な設定（不要な特権実行、不要なパッケージの同梱、秘密情報の埋め込み等）	Dockerは「rootless modeにより、デーモンとコンテナを非rootユーザーで実行し、デーモンおよびコンテナランタイムの潜在的脆弱性を緩和できる」と公式に説明している
IaC（Terraform等）	設定の誤り（misconfiguration）。公開範囲・保存時の暗号化・付与する権限の妥当性	TrivyはIaCの設定不備を公式スキャナのひとつとして備える。Checkov・tfsec等の専用ツールも併用候補となる（各ツールの仕様は導入時に公式ドキュメントで確認すること）
結果の集約	検出結果を開発者が参照する画面に一元化する	code scanningはSARIF（オープン標準）に基づくデータを取り込めるため、第三者ツールの結果も同じアラート画面・同じ運用手順で扱える

重大な問題がある場合のデプロイ停止

検査ジョブの失敗をデプロイジョブの前提条件に置き、Environmentsの承認ゲートと組み合わせる。停止条件は重大度で定義し（P12参照）、停止時は「誰が解除を承認できるか」「解除の有効期限」 をあらかじめ定める。停止条件を定めずに検査だけを追加すると、警告が蓄積するのみで抑止力が働かない。

出典：Trivy公式リポジトリ（github.com/aquasecurity/trivy）、Docker Docs「Rootless mode」、GitHub Docs「About code scanning（third-party code scanning tools）」。
事実：各ツールの機能仕様は上記に基づく／提案：デプロイ停止条件および承認手順の設計、ならびにベースイメージのダイジェスト固定は本資料による。

検査を実行する基盤自体が、最も権限の集中した攻撃対象となる

ワークフローはリポジトリへの書き込み権限とシークレットへのアクセスを併せ持つ。ゆえに「権限の最小化」「信頼できないコードの取り扱い」「長期有効な資格情報の排除」の3点が設計要件となる。公式ガイドはこれらを明文で警告している。

① 権限の最小化（permissions）

公式は「**GITHUB_TOKEN** の既定権限をリポジトリ内容の読み取りのみに設定し、必要に応じてジョブ単位で権限を引き上げることが良い慣行」と明示する。

フォークからのPRでは GITHUB_TOKEN は読み取り専用に制限される。

② 外部Actionの固定

公式は「完全長のコミットSHAへのピン留めが、Actionを不変のリリースとして使う現時点で唯一の方法」と説明する。

タグやブランチ参照は後から差し替え可能であり、参照先の改ざんに追随してしまう。

③ OIDCによるクラウド認証

OpenID Connect経由で、AWS／Azure／GCP／HashiCorp Vault等に対し**長期キーなしで短命トークン**を発行できる。

トークンのクレーム（repository、ref、environment等）により、リポジトリ・ブランチ・環境単位で許可範囲を絞れる。

フォーク由来のPRとpull_request_target

・公式は「**pull_request_target** および **workflow_run** を、信頼できないプルリクエストのチェックアウトと併用すると、リポジトリをセキュリティ侵害に晒す」と警告する。これらは特権的なトリガーであり、既定ブランチと同じキャッシュを共有し、書き込み権限とシークレットへのアクセスを持ち得る
・信頼できない入力を **run:** に直接埋め込むとスクリプトインジェクションとなる。公式は中間環境変数（env:）経由での受け渡し、またはJavaScript Actionへの切り出しを推奨する
・ワークフローの変更自体をレビュー必須にするため、**.github/workflows** をCODEOWNERSに含める手法が公式に示されている

実行環境の分離（ランナー）

・GitHub-hostedランナーは「一時的でクリーンな隔離済み仮想マシン内でコードを実行するため、環境を永続的に侵害する方法はない」と公式に説明される
・一方、セルフホステッドランナーについては「**パブリックリポジトリではほぼ決して使用すべきでない**。誰でもPRを開いて環境を侵害しうるため」と強く警告されている
・やむを得ず利用する場合は、REST APIで作成し最大1ジョブ実行後に自動削除される **Just-in-time runner** の利用と、ネットワーク・認証情報の分離を前提とする

出典：GitHub Docs「Security hardening for GitHub Actions」（permissions／untrusted code checkout／script injection／self-hosted runners／GitHub-hosted runners／CODEOWNERSによる監視）、「Automatic token authentication」「OpenID Connect」「Events that trigger workflows（pull_request）」。事実：いずれも公式ドキュメントの記述に基づく。

すべてを一律にブロックせず、重大度と新規性で判定を分ける

全件ブロックは開発の停止を、全件警告は形骸化を招く。判定軸を「重大度」と「新規／既存」に分け、
新規分の流入のみを確実に止めることが、実効性と開発速度を両立させる。

重大度	新規に持ち込まれる検出	既存の検出（導入時点の残存分）
Critical	マージをブロック （例外は責任者承認＋期限必須）	期限を定めた是正計画の対象。期限内に修正できない資産は隔離・撤去を検討
High	マージをブロック	是正計画の対象。優先度は稼働環境への露出度で判断
Medium	警告として表示し、次回対応の候補として記録	定期レビューで棚卸し。件数の推移を測定対象とする
Low	記録のみ。ブロックしない	記録のみ。件数の推移を測定対象とする

例外承認と誤検知の管理

・例外は「**承認者・理由・有効期限**」の3点を必ず記録する。期限のない例外は恒久化するため認めない
・誤検知はアラートを閉じるだけで終えず、判断理由と再評価予定日を残す。修正が反映された場合、code scanningのアラートは自動的にクローズされる
・Dependabotには、GitHubがキュレートした既定の自動トリアージ規則があり、誤検知に相当するアラートを相当程度自動的に除外できると公式に説明されている
・回避操作の可視化：push protectionの回避時には、アラート・監査ログ・所有者およびセキュリティマネージャーへの通知が生成される

セキュリティゲートの段階的厳格化

第1段階：可視化のみ
検査は実行するがブロックしない。検出件数の規模と誤検知率を把握し、対応体制を決める

第2段階：新規のみブロック
既存分は是正計画へ回し、新規に持ち込まれるCritical／Highのみ必須チェックで止める

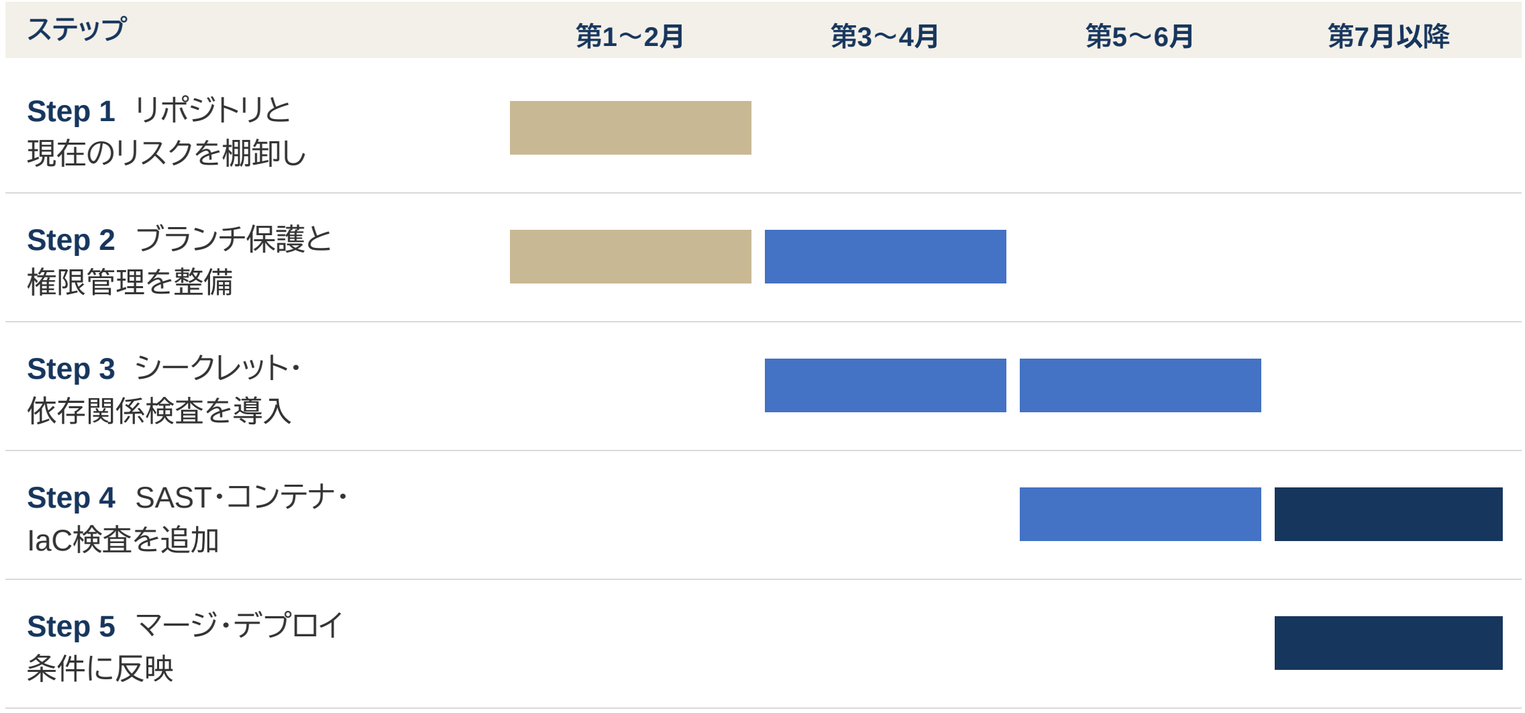
第3段階：対象範囲を拡大
コンテナ・IaC検査を必須化し、デプロイ前ゲートを追加。対象リポジトリを順次拡大する

第4段階：組織標準として固定
rulesetsで組織横断に適用し、例外の棚卸しと基準の定期レビューを制度化する

出典：GitHub Docs「Dependency review」「About code scanning」「Push protection」「GitHub security features」、CISA BOD 22-01（期限内に修正できない資産の隔離・撤去の要件）。
提案：重大度別の判定表および4段階の厳格化計画は本資料による設計案であり、公式が定める基準ではない。重大度の区分定義はFIRSTのCVSSに準拠して各組織で定義すること。

統制の整備を先行させ、検査は誤検知の少ない領域から順に追加する

導入順序は「棚卸し→統制整備→誤検知の少ない検査→解析系検査→判定への反映」とする。この順序であれば、開発者が最初に受け取る通知が処理可能な件数に収まり、検査の形骸化を避けられる。



役割分担と相談ルート

開発部門
検出内容の一次判断と修正。ワークフロー変更のPR提出。

運用部門
デプロイ前ゲートの承認、Environmentsの保護ルールと本番資格情報の管理。

セキュリティ部門
判定基準（重大度別のブロック条件）の策定、例外承認、基準の定期レビュー。

相談ルート
判断に迷う検出・例外申請の窓口を単一の経路に定め、応答期限を明示する。

意思決定事項
① 対象リポジトリの範囲と優先順位　② 重大度別のブロック条件と例外承認者　③ 検出対応の期限とその測定方法　④ プライベートリポジトリで必要となるライセンスの調達方針

出典：GitHub Docs「GitHub security features（提供条件）」「Security hardening for GitHub Actions」。事実：プライベートリポジトリでcode scanningにGitHub Code Security、secret scanningにGitHub Secret Protectionが必要である点は公式記述。
仮説：上記の月次スケジュールは標準的な導入規模を想定した例示（仮定値）であり、対象リポジトリ数と体制により変動する。提案：役割分担および意思決定事項は本資料による。

失敗の多くは技術ではなく、導入の順序と基準の欠落に起因する

検査を一度に導入すれば通知が処理能力を超え、判定基準がなければ例外が恒久化する。いずれも「段階導入」「基準の明文化」「定期レビュー」で回避できる。技術的な失敗として最も重いのは、外部Actionへの無条件な依存である。

よくある失敗	生じる帰結	改善策
検査を一度に全て導入し、大量のアラートが発生する	対応の優先順位が判断できず、既存分と新規分が混在したまま滞留する	まず可視化のみで規模を把握し、新規検出のCritical／Highからブロック対象とする段階導入をとる（P12参照）
誤検知が放置され、チェックが形骸化する	失敗したチェックを無視する運用が定着し、真の検出も見落とされる	判断理由と再評価予定日を記録して閉じる。Dependabotの既定の自動トリアージ規則を活用し、対象を絞り込む
ワークフローに過剰な権限を与える	侵害時の影響範囲がリポジトリ書き込みやシークレット全体に及ぶ	GITHUB_TOKENの既定を読み取りのみとし、必要なジョブ単位でpermissionsを引き上げる（公式が推奨する慣行）
外部Actionを無条件に利用する	参照先の改ざんにそのまま追随し、シークレットが窃取され得る	完全長のコミットSHAへピン留めする。Organization／リポジトリ単位でピン留めを強制するポリシーも設定する
検査時間が長くなり、開発者が回避し始める	検査のスキップや例外申請が常態化し、ゲートが機能しなくなる	PR時は差分中心の検査に限定し、全体解析はscheduleによる定期実行へ分離する（P7参照）
例外対応が恒久化する	一時的な承認が既定状態となり、リスクが台帳から消える	例外に有効期限を必須とし、期限到来時に再審査へ回す。棚卸しを定期レビューの議題に固定する

参考事例：CI/CDツール経由での資格情報流出（Codecov Bash Uploader、2021年）

攻撃者がアップローダスクリプトを改変し、CI環境変数（資格情報・トークン・キー）とgitのリモート情報を外部サーバーへ送出した。侵害期間は2021年1月31日～4月1日。
Codecovは4月1日にスクリプトを修正し、4月15日に影響を受けた利用者へ通知。同社は対処として「CI環境で env コマンドにより列挙される資格情報の再発行」を挙げている。

出典：Codecov公式セキュリティアップデート声明（about.codecov.io/security-update/）、GitHub Docs「Security hardening for GitHub Actions」「GitHub security features」。事実：Codecov事案の経緯・期間・推奨対処は同社公式声明に基づく／解釈・提案：失敗と改善策の対応付けは本資料による整理。なお「アラート疲れ」を定量的に示す公式統計は本調査では確認できていない。

「後から確認するセキュリティ」から「開発工程に組み込むセキュリティ」へ

検査をワークフローとしてコード化し、マージとデプロイの判定条件に組み込むことで、確認は工程の一部となり属人化を離れる。
導入の成否は技術選定ではなく、判定基準・例外管理・測定指標という運用設計が定義されているかで決まる。

Ⅰ. 技術的な整備項目

- □ ブランチ保護とレビューの必須化（直接pushの禁止）
- □ ワークフロー権限の最小化（GITHUB_TOKENは読み取り既定）
- □ 外部ActionのコミットSHAによる固定
- □ SASTの自動実行（code scanning：PR時＋定期実行）
- □ 依存ライブラリの脆弱性検査（Dependabot alerts／dependency review）
- □ シークレット漏えい対策（secret scanning／push protection）
- □ コンテナイメージとIaCの検査（SARIFでcode scanningへ集約）
- □ クラウド認証のOIDC化（長期有効なアクセスキーを保管しない）

Ⅱ. 運用面の整備項目

- □ 重大度に応じたセキュリティゲートの定義（何を止め、何を通すか）
- □ 新規検出と既存検出の区別、既存分の是正計画と期限
- □ 例外の記録（承認者・理由・有効期限の3点を必須とする）
- □ 誤検知の判断理由と再評価予定日の記録
- □ 開発・運用・セキュリティ部門の役割分担と相談窓口の明示
- □ 検出数・対応時間・再発率の継続測定
- □ 判定基準の定期レビューと段階的な厳格化
- □ プライベートリポジトリで必要なライセンスの確認

セキュリティ確認を工程の終端に置く限り、手戻りは構造的に発生する。判定条件をコードとして共有することが、継続的な安全性の前提となる。

出典：GitHub Docs（About code scanning／Dependency review／About secret scanning／Push protection／Security hardening for GitHub Actions／OpenID Connect／About protected branches／About rulesets／GitHub security features）、
OWASP DevSecOps Guideline、CISA BOD 22-01。提案：本チェックリストの項目構成および運用面の要件は本資料による整理であり、各組織の統制要件に応じた調整を前提とする。

DevSecOpsの導入について、お気軽にご相談ください

本資料で示した導入ステップは、対象リポジトリの規模・体制・既存の統制状況によって最適な順序が変わる。
現状の棚卸しから判定基準の設計、段階導入の計画づくりまで、実行を前提としたご支援を行っている。

お問い合わせ先

Start Challenge Consulting
挑む力が、世界を照らす

メール
info@startchg.com

担当
平出

ご相談いただける内容

- 現状のリポジトリ統制とリスクの棚卸し（P6・P13）
- ブランチ保護・権限設計とrulesetsへの移行方針（P6）
- 検査ワークフローの設計と段階導入計画の策定（P5・P13）
- 重大度別のマージ／デプロイ判定基準の設計（P12）
- 例外承認・誤検知の運用ルールと台帳の整備（P12）
- OIDC移行を含むワークフロー堅牢化の点検（P11）
- プライベートリポジトリで必要なライセンスの整理（P13）
- 開発・運用・セキュリティ部門の役割分担の設計（P13）

まずは現状のリポジトリ構成と統制状況をお知らせください。検査の追加順序と判定基準の初期案をご提示します。

本資料の記載内容は、GitHub公式ドキュメントおよび各ツール公式ドキュメントの記述（取得時点）に基づく。機能の提供条件はプランおよび改訂により変動するため、導入時に各公式ページで再確認すること。
括弧内のページ番号は本資料の該当ページを示す。