PGECons
PostgreSQL Enterprise Consortium

PostgreSQL エンタープライズ・コンソーシアム 技術部会 WG#3

PostgreSQL セキュリティガイド

改訂履歴

版	改訂日	変更内容
1.0	2015/04/23	初版
2.0	2016/04/18	2015 年度成果を追記 • ４章 PCI DSS への PostgreSQL 対応調査結果一覧に以下の項目を追記 ◦ 「オブジェクト監査」 ◦ 「（参考）今後、監査のユースケースに求められる PostgreSQL の強化機能について」 ◦ 「パスワードポリシーの設定」 ◦ 「パスワードの総当り攻撃対策」 ◦ 「行単位のアクセス制御」 ◦ 「SQL レベルのファイアウォール制御」 ◦ 「格納データの透過的暗号化」 • ５章「検証結果」を追記 ◦ 「透過的暗号化」 ◦ 「行単位のアクセス制御」 ◦ 「PostgreSQL サーバログへの監査データ出力による性能および運用性の検証」 • そのほか一部の情報を 2016 年 3 月現在の情報に修正

本資料について

■PostgreSQL エンタープライズコンソーシアムと WG3 について

エンタープライズ領域における PostgreSQL の普及を目的として設立された PostgreSQL エンタープライズコンソーシアム（以降 PGECons）では、技術部会における PostgreSQL の普及に対する課題の検討を通じて 2014 年度および 2015 年度の活動テーマを挙げ、その中から 3 つのワーキンググループで具体的な活動を行っています。

- WG1（性能ワーキンググループ）
- WG2（移行ワーキンググループ）
- WG3（設計運用ワーキンググループ）

WG3 では、ミッションクリティカル性の高いエンタープライズ領域で必要とされる DBMS の非機能要求に着目し、PostgreSQL の典型的システム方式の調査および動作検証を行い、技術ノウハウを整理してきました。2014 年度は可用性、セキュリティの観点に着目して活動を実施し、2015 年度は継続してセキュリティの観点に向けた活動と、新しい観点として運用性を高めるツールに向けた活動を行っています。

■本資料の概要と目的

本資料は 2014 年度および 2015 年度の WG3 における活動として、PostgreSQL を対象としたデータベースセキュリティの調査と対策手段についてまとめたものです。

■本資料の構成

1 章 はじめに

本資料を読む前にご確認いただきたいことを記載しています。

2 章 企業システムで要求されるセキュリティ要件

「PostgreSQL は企業システムで安心して使えるか」を評価するには、企業システムに求められるセキュリティ要件が明確でなければなりません。そこで、本資料では最初に企業システムに求められるセキュリティ要件について整理します。そして、クレジットカード業界のセキュリティ標準である PCI DSS を紹介し、データベースセキュリティコンソーシアム(DBSC)による PCI DSS 対応表を用いることで、DBMS に求められるセキュリティ要件を整理します。

3 章 PCI DSS への PostgreSQL 対応調査結果一覧

2 章で紹介した DBSC による PCI DSS 対応表に対して、PostgreSQL の対応状況を加えた調査結果を一覧にまとめたものです。

4 章 調査結果詳細

3 章でまとめた調査結果の各項目について、調査結果の詳細、設計手順・運用手順を説明します。

5 章 検証結果

3 章でまとめた調査結果の項目のなかで、性能や運用性上の留意点を確認するために実施した検証結果を記載しています。

6 章 おわりに

本資料の総括をします。

■想定読者

本資料の読者は以下のような知識を有していることを想定しています。

- DBMS を操作してデータベースの構築、保守、運用を行うデータベース管理者の知識
- PostgreSQL を利用する上での基礎的な知識
- PCI DSS の要件チェックのチェック表として利用

■資料内の記述について

本資料の記法については、必要に応じて各章の冒頭に記載しています。

■謝辞

本資料を作成するにあたって、データベース・セキュリティ・コンソーシアム（DataBase Security Consortium）で作成公開された成果物を参考にし、また一部転記いたしました。この場を借りて厚く御礼を申し上げます。

目次

1. はじめに

昨今、情報システムで用いられている顧客情報などの機密情報の漏えい事件が数多く起きており、社会全体として、情報システムへの頑強なセキュリティ対策が必要とされてきています。特に機密情報を格納しているデータベース（以下 DB）で、十分なセキュリティ対策が情報システムにおいては必須の要件となっています。

本資料は WG3 におけるセキュリティ検討の活動成果を報告するもので。

PostgreSQL でどの程度セキュリティ対策ができるかを検討するためにクレジットカード業界のセキュリティ標準である Payment Card Industry Data Security Standards(以下、PCI DSS)[1]に照らし合わせて調査した結果をまとめています。

本資料では、PCI DSS で要求されている各項目に対して、PostgreSQL で実現できる手法や考え方について記載しています。各セキュリティ対策への対応可否や PostgreSQL を利用した PCI DSS 対応の環境を設計・運用する上での参考資料として利用してください。

2015 年 4 月に公開した第一版では机上検証を中心に、成果を取りまとめました。2016 年 4 月に公開する第二版では、技術検証の対象を広げています。また、実際にセキュリティを向上させるために必要な動作による影響を調査するため、暗号化や監査については実機での検証を行い、留意するポイントの確認を行っています。

なお、本資料で記載した PostgreSQL での各手法については、性能や運用性などデータベースシステムに与える要素を考慮したものではなく、セキュリティ対策に対する実現性の有無についてを主眼に置き、記載を行っています。本書の内容を適用される際には、事前にシステム全体への影響範囲などを確認の上ご利用いただくことを強くお勧めします。

データベース・セキュリティ・コンソーシアム(DataBase Security Consortium、以下 DBSC)では、PCI DSS を基準にした場合において、データベースが実施すべき要件事項をまとめた成果物が作成公開されています。本資料を作成するにあたって、これらの成果物を参考にしており、転記部分については、DBSC が著作権を保有しています。

【出典】「データベースセキュリティガイドライン(2.0 版)」
データベース・セキュリティ・コンソーシアム（DBSC）
http://www.db-security.org/

2016 年 1 月現在、PCI DSS の最新版は v3.1 となっておりますが、本資料では DBSC が公開している最新のガイドラインに合わせて PCI DSS v1.2 に対応しています。また、PostgreSQL に特化した内容を記載しています。一般的な RDBMS のセキュリティ対策に関しては、DBSC のガイドラインをご参照ください。

本資料では、特に明記がない限り PostgreSQL 9.3 を対象としています。また、紹介するセキュリティ対策の中には、OS の機能を利用するものがあります。その場合は同様に特に明記がない限り、Redhat Enterprise Linux 6.5 を想定環境としています。

1 PCI Security Standards Council
PCI DSS https://ja.pcisecuritystandards.org/minisite/en/

2. 企業システムで要求されるセキュリティ要件

「PostgreSQL は企業システムで安心して使えるか」を評価するには、企業システムに求められるセキュリティ要件が明確でなければなりません。そこで、本資料では最初に企業システムに求められるセキュリティ要件について整理します。そして、クレジットカード業界のセキュリティ標準である PCI DSS を紹介し、データベースセキュリティコンソーシアム(DBSC)による PCI DSS v1.2 対応表を用いることで、データベース管理システム（以下 DBMS）に求められるセキュリティ要件を整理します。

2.1. 非機能要求とセキュリティ

情報システムは「業務アプリケーション」と「システム基盤」の大きく二つの要素より構成されています。

「業務アプリケーション」はビジネス・業務そのものの機能を実現する仕組みであり、これらに対する要求事項が「機能要求」となります。一方、「システム基盤」は業務アプリケーションを実行するためのインフラであり、これらシステム基盤に対する要求事項が業務機能と区別して「非機能要求」と整理されています。

ここで「機能要求」は情報システムで実現したいビジネスそのものを具現化した機能に対する要求のことであり、そのビジネスの専門家（ユーザ）と IT 技術者（ベンダ）が協力して「業務アプリケーション」の設計に反映していくべき項目となります。

一方、「非機能要求」は情報システムのシステム基盤に対する要求ですが、IT の専門知識が豊富ではないビジネスの専門家（ユーザ）が適切に要求事項の整理を行うことは一般的には難しいと考えられます。また、開発対象の情報システムにおけるビジネスの知識や経験が浅い IT 技術者（ベンダ）にとっても、ユーザに最適な要求条件を適切なタイミングで提示することはきわめて困難であり、「システム基盤」構築にあたっては様々なリスクが生じることが実態です。

このビジネスの専門家（ユーザ）と IT 技術者（ベンダ）間で必要な「非機能要求」に対する共通認識を持つことがとても重要であり、事前に両者で合意しておかなくてはならない項目について、独立行政法人 情報処理推進機構（IPA）にて「非機能要求グレード利用ガイド」として整理が行われています[2]。

この「非機能要求グレード」には、「可用性」「性能・拡張性」「運用・保守性」「移行性」「セキュリティ」「システム環境・エコロジー」の大項目があります。(表 2.1)ここで今回の目的である、情報システムの安全性を確保したいという要求に応えるためには、「セキュリティ」に着目することが重要であると考えられます。

表 2.1: 非機能要求とは

大項目	概要	要求例
可用性	システムサービスを継続的に利用可能とするための要求	・運用スケジュール（稼働時間・停止予定など） ・障害、災害時における稼働目標
性能・拡張性	システムの性能および将来のシステム拡張に対する要求	・業務量および今後の増加見積もり ・システム化対象業務の特性（ピーク時、通常時、縮退時）
運用・保守性	システム運用と保守サービスに関する要求	・運用中に求められるシステム稼働レベル ・問題発生時の対応レベル
移行性	現行システム資産の移行に関する要求	・新システムへの移行期間および移行方法 ・移行対象資産の種類および移行量
セキュリティ	情報システムの安全性の確保に関する要求	・利用制限 ・不正アクセスの防止
システム環境・エコロジー	システムの設置環境やエコロジーに関する要求	・耐震/免震、重量/空間、温度/湿度、騒音などシステム環境に関する事項 ・CO2 排出量や消費エネルギーなどエコロジーに関する事項

2 独立行政法人情報処理推進機構
非機能要求グレード http://www.ipa.go.jp/sec/softwareengineering/reports/20100416.html

2.2. DBSC による PCI DSS 対応表

DBSC では、データベース・セキュリティの普及促進を図る活動の一環として、データベース・セキュリティに関連する資料を作成公開しています。成果物には、一般的な RDBMS に求められるセキュリティ要件を整理した資料に加えて、PCI DSS の要件に対応する DB セキュリティ対策をまとめた資料も含まれます。

PCI DSS とは、クレジットカードの会員データのセキュリティを強化し、均一なデータセキュリティ評価基準の採用をグローバルに推進するために策定された、クレジットカード業界のセキュリティ標準です。DBSC による「PCI DSS データベースセキュリティガイドライン 追補版 第 1.0 版」では、PCI DSS v1.2 の要件事項でデータベースに関連する要件事項に対して、具体的なデータベース・セキュリティ対策が記載されています。

本資料では以下の DBSC の成果物および PCI DSS のドキュメントを引用、参考にしています。

- データベース・セキュリティ・コンソーシアム (2009), データベースセキュリティガイドライン 第 2.0 版.
- データベース・セキュリティ・コンソーシアム (2009), PCI DSS データベースセキュリティガイドライン 追補版 第 1.0 版.
- データベース・セキュリティ・コンソーシアム (2009), DB セキュリティ 製品別機能対応表 インストール・初期設定編 第 1.0 版.
- データベース・セキュリティ・コンソーシアム (2009), DB セキュリティ 製品別機能対応表 検知・追跡系編 第 1.0 版.
- PCI Security Standards Council (2008), Payment Card Industry(PCI) Data Security Standard – Requirements and Security Assessment Procedures, Version 1.2.[3]

本資料では、DBSC がまとめている一般的な RDBMS におけるデータベースセキュリティガイドラインの考え方に従って、PostgreSQL に当てはめた場合について記載しています。3 章で DBSC の成果物を参考に、PostgreSQL を PCI DSS の要件に照らし合わせることで、どの程度 PostgreSQL でセキュリティ対策が出来るかを把握し、4 章で PCI DSS を満たすにはどのような対策が必要になるかについて調査・検証した結果をまとめています。

3 https://www.pcisecuritystandards.org/documents/PCI_Security_Assessment_Procedures_v1-2.pdf

3. PCI DSS への PostgreSQL 対応調査結果一覧

本章では、PCI DSS 要件を満たす必要のあるシステムに対し、PostgreSQL がデータベースとして採用された場合に必要となるセキュリティ要件の対応可否判断と対応策の概要について記載しています。表 3.1 に記載項目の説明、表 3.2 に調査結果を整理します。

データベースが必要とするセキュリティ要件には、DataBase Security Consortium(以降 DBSC)によって公開されている、PCI DSS を基準にした場合におけるデータベースが実施すべき要件と具体的なセキュリティ対策が記載された資料[4]から引用しています。

なお本章の脚注のリンクは PostgreSQL 9.5 をベースとしています。

表 3.1: 「表 3.2: PCIDSS 要件への PostgreSQL 対応」の項目に関する説明

項目		内容	備考
#		PCI DSS の項目番号	DBSC の資料から抜粋しています。
PCI DSS 要件		PCI DSS に記載されている要件	
DBSC 対応要件		DBSC によるデータベースセキュリティ要件	
PostgreSQL 対応	対応レベル	○：対応可能 △：条件付きで可能 ×：対応不可 －：ソフトウェア機能は無関係	PGECons が記載した箇所です。 ※「PostgreSQL 対応」とは、PostgreSQL 9.3、Red Hat Enterprise Linux 6、PostgreSQL に関連する OSS および運用で対応できることを示しています。
	対応策	対応策の概要です。 詳細な説明が必要な場合は第 4 章に別途記載しています。	

「PostgreSQL 対応」には対応レベルと対応策を記載しています。対応レベルは、要件事項を実現するにあたり PostgreSQL の標準機能で実現できる場合に「○」、一部だけ実現できる、あるいは、外部ツールや OS 機能を組み合わせれば実現できる場合に「△」、対応できない場合に「×」を記載しています。データベースソフトウェア機能の有無とは関係ない項目と、項目番号の枝番で詳細を記述していて、大番号には概略を記載している場合（例:「要件 3 保存されたカード会員データを保護すること」）には、対応レベルを「－」としています。

対応策について詳細な手順を記載する必要があると判断した項目は、第 4 章に別途記載しています。

表 3.2: PCIDSS 要件への PostgreSQL 対応

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
要件:1	カード会員データを保護するために、ファイアウォールをインストールして構成を維持すること	・DB サーバーに接続できるノードは必要最小限にする。	○	• ファイアウォールを導入した環境で運用可 • postgresql.conf の listen_addresses や pg_hba.conf の認証設定で対応可
要件: 2	システムパスワードおよび他のセキュリティパラメータにベンダ提供のデフォルト値を使用しないこと		－	－
2.1	システムをネットワーク上に導入する前に、ベンダ提供のデフォルト値を必ず変更する（パスワード、簡易ネット	・デフォルトポートを変更する。 ・デフォルトで作成される不要なアカウントを削除または無効化する。	○	• postgresql.conf の変更や ALTER ROLE 等で対応可

4 DataBase Security Consortium PCI DSS データベースセキュリティガイドライン 追補版 http://www.db-security.org/

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
	ワーク管理プロトコル（SNMP）コミュニティ文字列の変更、不必要なアカウントの削除など）。	・デフォルトパスワードを変更する。 ・サンプル DB を削除する。 ・パブリックなロールから不要な権限を削除する。		
2.2	すべてのシステムコンポーネントについて、構成基準を作成する。この基準は、すべての既知のセキュリティ脆弱性をカバーし、また業界で認知されたシステム強化基準と一致している必要がある。		−	−
2.2.1	1 つのサーバには、主要機能を 1 つだけ実装する。	・DB サーバでは、主として DBMS のみを稼動させる。	○	• そのような運用を行うことで対応可
2.2.2	安全性の低い不必要なサービスおよびプロトコルはすべて無効にする（デバイスの特定機能を実行するのに直接必要でないサービスおよびプロトコル）。	・必要な機能を選択し、対象機能のみをインストールする。 ・使用しない DBMS 機能、サービスは削除または無効化する。 ・DB サーバの OS レベルにおいても、不要な機能、サービスを削除する。 ・必要なプロトコルのみを有効にし、不要なプロトコルは無効にする。	△	• RPM でインストールする要素の取捨選択は可能 • RPM でインストールする場合は、contrib モジュールについては、その中の一部のみインストールは不可 • ソースのコンパイルによるインストールの場合は、contrib モジュール単位での選択が可能 • OS 依存のものは、そのような運用で対応可
2.2.3	システムの誤用を防止するためにシステムセキュリティパラメータを構成する。	・各種ベンダから提供される推奨パラメータを設定する。 例） ・データベースセキュリティガイドライン 第 2.0 版 http://www.db-security.org/report.html#gl02 ・DB セキュリティ 製品別機能対応表 第 1.0 版 http://www.db-security.org/wgimpl_seika.html	○	• 商用の監査ツールにおける PostgreSQL の対応状況およびその機能詳細については以下文書にて詳細を解説： [4.7（参考）今後、監査のユースケースに求められる PostgreSQL の強化機能について]
2.2.4	スクリプト、ドライバ、機能、サブシステム、ファイルシステム、不要な Web サーバなど、不要な機能をすべて削除する。	・使用しない DBMS 機能、オブジェクトなどは削除または無効化する。 ・DB サーバの OS レベルにおいても、不要な機能、サービスを削除する。	△	• RPM では contrib モジュール群全体が一つのパッケージになっていて、一部だけ削除はできない • ソースのコンパイルによるインストールの場合は、contrib モジュール単位での選択が可能 • OS 依存のものは、そのような運用で対応可
2.3	すべてのコンソール以外の管理アクセスを暗号化する。Web ベースの管理やその他のコンソール以外の管理	・DBMS の暗号化通信機能を有効にする、あるいはその他の方法で通信を暗号化して DB クライアントからアクセスする。	○	• SSL 接続機能にて対応可 PostgreSQL マニュアル以下項を参照： [17.9 SSL による安全な TCP/IP 接続] [5]

5 http://www.postgresql.jp/document/9.5/html/ssl-tcp.html
・データベースセキュリティガイドライン 第 2.0 版 http://www.db-security.org/report.html#gl02

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
	アクセスについては、SSH、VPN、またはSSL/TLS などのテクノロジを使用する。			
要件：3	保存されたカード会員データを保護すること。		－	－
3.4	以下の手法を使用して、すべての保存場所で PAN[6] を少なくとも読み取り不能にする（ポータブルデジタルメディア、バックアップメディア、ログを含む）。 ・強力な暗号化技術をベースにしたワンウェイハッシュ・トランケーション ・インデックストークンとパッド（パッドは安全保存する必要がある） ・関連するキー管理プロセスおよび手順を伴う、強力な暗号化アカウント情報のうち、少なくとも PAN は読み取り不能にする必要がある。 注: 何らかの理由で PAN を読み取り不能にできない場合は、「付録 B:代替コントロール」を参照。 強力な暗号化技術は、「PCI DSS Glossary of Terms, Abbreviations, and Acronyms」で定義されています。		－	• （3.4.1 項目に記載しています）
3.4.1	（ファイルまたは列レベルのデータベース暗号化ではなく）ディスク暗号化が使用される場合、論理アクセスはネイティブなオペレーティングシステムのアクセス制御メカニズムとは別に管理する必要がある（ローカルユーザーアカウントデータベースを使用しないなどの方法で）。暗号解除キーをユーザアカウントに結合させてはいけない。	・データ暗号化機能／ツールを利用し、格納データを暗号化する。 ・データ暗号化機能／ツールを利用し、データファイルを暗号化する。 ・データ暗号化機能／ツールを利用し、バックアップファイルを暗号化する。	△	• pgcrypto により格納データの暗号化が可 • 透過的暗号化(TDE)を実装する事で透過的な暗号化および復号化が可能 • その他については OS 機能、外部ツールの機能を利用して実現する 以下文書にて詳細を解説: [4.29 格納データの暗号化] [4.30 格納データの透過的暗号化 [4.31 ファイルシステム透過的暗号化]
3.5	カード会員データの暗号化に使用される暗号化キーを、漏洩と誤使用から保護する。	・適切に暗号鍵を管理する。 例) ・定期的に鍵の暗号鍵を変更す	△	• 本表 #3.4.1 に記載の各手法で暗号化を行う場合に、その鍵の管理について左記

・DB セキュリティ 製品別機能対応表 第 1.0 版　http://www.db-security.org/wgimpl_seika.html

6　PAN(Primary Account Number):カード会員番号

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
		る。		要件にしたがった運用をすることで対応可
3.5.1	暗号化キーへのアクセスを、必要最小限の管理者に制限する。	・暗号鍵にアクセス可能な人物を、最小限の DB 管理者に限定する。	△	
3.5.2	暗号化キーの保存場所と形式を最小限にし、安全に保存する。	・暗号鍵を暗号化して保存する。 ・鍵の暗号鍵とデータの暗号鍵を分けて保管する。	△	
3.6	カード会員データの暗号化に使用されるキーの管理プロセスおよび手順をすべて文書化し、実装する。これには、以下が含まれる。	・暗号鍵のライフサイクル（生成、配布、保存、廃棄など）ごとに適切に管理する。	△	
要件：4	オープンな公共ネットワーク経由でカード会員データを伝送する場合、暗号化すること		―	―
要件：5	アンチウィルスソフトウェアまたはプログラムを使用し、定期的に更新すること		―	―
要件：6	安全性の高いシステムとアプリケーションを開発し、保守すること		―	―
6.1	すべてのシステムコンポーネントとソフトウェアに、ベンダ提供の最新セキュリティパッチを適用する。重要なセキュリティパッチは、リリース後 1 カ月以内にインストールする。 注: 組織は、パッチインストールの優先順位を付けるために、リスクに基づくアプローチの適用を検討できる。たとえば、重要なインフラストラクチャ（一般に公開されているデバイス、システム、データベースなど）に重要性の低い内部デバイスよりも高い優先順位を付けることで、優先順位の高いシステムおよびデバイスは 1 カ月以内に対処し、重要性の低いシステムおよびデバイスは 3 カ月以内に対処するようにする。	・最新の DBMS のセキュリティパッチを適用する。	○	• マイナーバージョンアップを適用することで対応可 • PostgreSQL では公式なリリースとしての「セキュリティパッチ」は無く、マイナーバージョンアップとして提供される （重大なセキュリティ脆弱性が確認された場合は、早急なマイナーバージョンアップがなされることが通例となっている）
6.2	新たに発見された脆弱性を特定するためのプロセスを確立する（インターネット上	・DB に関するセキュリティの事故や脆弱性の最新情報を常に収集する。	○	• コミュニティにより情報が開示されています。

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
	で無料で入手可能な警告サービスに加入するなど)。新たな脆弱性の問題に対処するために、PCI DSS 要件 2.2 で要求されているとおりに構成基準を更新する。			
6.3	PCI DSS (安全な認証やロギングなど)に従い、業界のベストプラクティスに基づいてソフトウェアアプリケーションを開発し、ソフトウェア開発ライフサイクル全体を通して情報セキュリティを実現する。これらのプロセスには、以下を含める必要がある。		−	−
6.3.1	導入前にすべてのセキュリティパッチ、システムとソフトウェア構成の変更をテストする(以下のテストが含まれるが、これらに限定されない)。		−	−
6.3.1.3	暗号化による安全な保存の検証	・データ暗号化機能／ツールを利用し、格納データを暗号化する。	○	• pgcrypto により格納データ/データファイルの暗号化が可 • その他については外部ツールや OS の機能を利用して実現する 以下文書に詳細を解説: [4.29 格納データの暗号化] [4.30 格納データの透過的暗号化] [4.31 ファイルシステム透過的暗号化]
6.3.1.5	適切な役割ベースのアクセス制御(RBAC)の検証	・ロール、または各アカウント別に、最低でもテーブル、可能であれば列(カラム)単位でアクセス制御する。	○	• 列単位でのアクセス制御可
6.3.2	開発/テスト環境と本番環境の分離	・開発環境と本番環境は、インスタンスを分離する、可能であれば、異なるノードに作成する。	○	• 分離することは可能である
6.3.5	本番環境システムがアクティブになる前にテストデータとテストアカウントを削除する	・DB 内のテストデータ、テスト用 DB アカウントを削除する。	○	• 削除することは可能である
6.3.6	アプリケーションがアクティブになる前、または顧客にリリースされる前に、カスタムアプリケーションアカウント、ユーザー ID、パスワードを削除する	・DB 内のカスタムアプリケーション用 DB アカウントを削除する。	○	• 削除することは可能である
要件: 7	カード会員データへのアクセスを、業務上必要な範囲		−	−

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
	内に制限すること			
7.1	システムコンポーネントとカード会員データへのアクセスを、業務上必要な人に限定する。アクセス制限には以下を含める必要がある。		−	−
7.1.1	特権ユーザー ID に関するアクセス権が、職務の実行に必要な最小限の特権に制限されていること	・管理者用 DB アカウントについて、DB へのアクセス要件に基づき、必要最低限のアクセス権限を付与する。 ・管理者権限は、DB アカウントを限定して付与する。	△	• 各管理者用アカウントに対し、接続できるデータベースを限定する設定が可能 • ユーザにスーパーユーザ権限の一部だけを与えるということができない
7.1.2	特権の付与は、個人の職種と職能に基づくこと	・役割ごと、利用者ごとに、適切な権限を持ったアカウントを整理する。 ・DB 管理者アカウントは特定の者しか、利用できないようにする。 ・DB 管理者アカウントを担当者別に割当てる。	○	• ROLE 設定により対応可 PostgreSQL マニュアル以下項を参照: [20 章 データベースロール][7]
7.1.4	自動アクセス制御システムを実装する	・DBMS のアクセス制御機能により、DB アカウント（またはロール）別にテーブル単位（可能であれば列単位）にアクセス制御する。	○	• テーブル単位、列単位でのアクセス制御可 PostgreSQL マニュアル以下項を参照: [リファレンス SQL コマンド GRANT] [8]
7.2	複数のユーザーを持つシステムコンポーネントに対して、ユーザーの必要な範囲に基づいてアクセスを制限し、特に許可されていない限り「すべてを拒否」に設定した、アクセス制御システムを確立する。アクセス制御システムには以下を含める必要がある。		−	−
7.2.2	職種と職能に基づく、個人への特権の付与	・役割ごと、利用者ごとに、適切な権限を持ったアカウントを整理する。 ・DB 管理者アカウントは特定の者しか、利用できないようにする。 ・DB 管理者アカウントを担当者別に割当てる。	○	• ROLE 設定により対応可 PostgreSQL マニュアル以下項を参照: [20 章 データベースロール][9]
要件：8	コンピュータにアクセスできる各ユーザに一意の ID を割り当てる。		−	−
8.1	システムコンポーネントまた	・DB アカウントを担当者別に割	○	• ROLE 設定により対応可

7 http://www.postgresql.jp/document/9.5/html/user-manag.html
8 http://www.postgresql.jp/document/9.5/html/sql-grant.html
9 http://www.postgresql.jp/document/9.5/html/user-manag.html

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
	はカード会員データへのアクセスを許可する前に、すべてのユーザに一意の ID を割り当てる。	当てる。(作成する)		
8.2	一意の ID の割り当てに加え、以下の方法の少なくとも 1 つを使用してすべてのユーザを認証する。 ・パスワードまたはパスフレーズ ・2 因子認証(トークンデバイス、スマートカード、生体認証、公開鍵など)	・DB アカウントの認証方式を左記のいずれかにする。	○	• パスワード、公開鍵による認証が可能 • 2 因子認証は PostgreSQL 機能としては対応していない
8.4	(「PCI DSS Glossary of Terms,Abbreviations, and Acronyms」で定義されている)強力な暗号化を使用して、すべてのシステムコンポーネントでの伝送および保存中にすべてのパスワードを読み取り不能にする。	・一般的な商用 DBMS なら問題なし	○	• SSL 接続機能を利用できる • 保存中データベースのパスワードはハッシュ格納され、ハッシュデータを任意に参照できるのは管理者ユーザのみ • アプリケーション上のユーザパスワードは pgcrypto により強力な各種暗号化にて格納可能 PostgreSQL マニュアル以下項を参照: [17.9 SSL による安全な TCP/IP 接続] [10] 以下の項にて詳細を解説: [4.29 格納データの暗号化]
8.5	すべてのシステムコンポーネントで、以下のように、消費者以外のユーザおよび管理者に対して適切なユーザ認証とパスワード管理を確実に行う。		−	−
8.5.3	初期パスワードをユーザごとに一意の値に設定し、初回使用後に直ちに変更する。	・DBMS の機能により、初期パスワードを強制的に変更させる設定で DB アカウントを作成する。	△	• 初回使用後にパスワードをただちに変更させる設定は無いが、外部認証を使えば可能 以下の項にて詳細を解説: [4.1 アカウントポリシー機能の実現]
8.5.4	契約終了したユーザのアクセスは直ちに取り消す。	・契約終了した担当者用の DB アカウントを削除する。	○	• そのように運用可能
8.5.5	少なくとも 90 日ごとに非アクティブのユーザアカウントを削除/無効化する。	・少なくとも 90 日ごとに不要な DB アカウントを削除または無効化する。	○	• そのように運用可能
8.5.6	リモート保守のためにベンダが使用するアカウントは、必要な期間のみ有効にする。	・一時的な利用者にはその都度、DB アカウントに対し一時的なパスワードを与える。	○	• そのように運用可能
8.5.8	グループ、共有、または汎用のアカウントおよびパスワードを使用しない。	・DB アカウントを担当者別に割当てる。(作成する)	○	• そのように運用可能

10 http://www.postgresql.jp/document/9.5/html/ssl-tcp.html
・表中「#8.5」に含まれる「以下」とは、PCI DSS 要件の 8.5.1～8.5.16 を指しています。この資料には記載されていません。

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
8.5.9	少なくとも 90 日ごとにユーザパスワードを変更する。	・少なくとも 90 日ごとに DB アカウントのパスワードを変更する。	○	• そのように運用可能 • 一定周期でパスワード変更を自動的に強制するには外部ソフトウェアとの連携が必要 以下の項にて詳細を解説: [4.1 アカウントポリシー機能の実現]
8.5.10	パスワードに 7 文字以上が含まれることを要求する。	・DBMS の機能により、パスワードの複雑性を設定する。	○	• passwordcheck(パスワード検査)機能や外部認証によるアカウントポリシー適用により、対応可 PostgreSQL マニュアル以下項を参照: [F.23 パスワード検査] [11] 以下の項にて詳細を解説: [4.1 アカウントポリシー機能の実現] [4.12 パスワードポリシーの設定]
8.5.11	数字と英文字の両方を含むパスワードを使用する。	・DBMS の機能により、パスワードの複雑性を設定する。	○	
8.5.12	ユーザが新しいパスワードを送信する際、最後に使用した 4 つのパスワードと同じものを使用できないようにする。	・DBMS の機能により、利用可能なパスワード履歴を設定する。	△	• 外部認証によるアカウントポリシー適用により、対応可 以下の項にて詳細を解説: [4.1 アカウントポリシー機能の実現] [4.13 パスワードの総当たり攻撃対策]
8.5.13	最大 6 回の試行後にユーザ ID をロックアウトして、アクセス試行の繰り返しを制限する。	・DBMS の機能により、ロックアウトを設定する。	△	
8.5.14	ロックアウトの期間を、最小 30 分または管理者がユーザ ID を有効にするまで、に設定する	・DBMS の機能により、ロックアウトを設定する。	△	
8.5.15	セッションが 15 分を超えてアイドル状態の場合、端末を再有効化するためにユーザにパスワードの再入力を要求する。	・DBMS の機能により、アイドル 21 時間を設定する。	△	• 該当する機能は無い • 接続プロキシソフトウェアを使う方法、監視スクリプトを使う方法等で対応可 以下の項にて詳細を解説: [4.33 長時間アイドル中の接続を自動切断する]
8.5.16	カード会員データを含むデータベースへのすべてのアクセスを認証する。これには、アプリケーション、管理者、およびその他のすべてのユーザによるアクセスが含まれる。	・一般利用者はアプリケーション経由でのみデータにアクセスできるようにし、アプリケーションは利用者を認証、認可を適切に実施する。 ・DB に直接アクセスできるのは、DB 管理者に限定する。	○	• pg_hba.conf 設定にてそのように設定可能 PostgreSQL マニュアル以下項を参照: [第 19 章クライアント認証] [12]
要件: 9	カード会員データへの物理アクセスを制限する。		−	−
要件:10	ネットワークリソースおよび		−	−

11 http://www.postgresql.jp/document/9.5/html/passwordcheck.html
12 http://www.postgresql.jp/document/9.5/html/client-authentication.html

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
	カード会員データへのすべてのアクセスを追跡および監視する。			
10.2	以下のイベントを再現するためにすべてのシステムコンポーネントの自動監査証跡を実装する。		−	• 10.2 以下の諸項目は PostgreSQL 標準機能による対応のほか、監査ログ拡張機能を提供する拡張モジュールや商用ソフトウェアも利用可能 以下の項にて詳細を解説: [4.2 pgaudit 拡張モジュールの使用] [4.3 オブジェクト監査] [4.4 PostgreSQL を拡張した商用製品による監査]
10.2.1	カード会員データへのすべての個人アクセス	・DBに対するすべてのアクセスを記録する。	○	• log_statement 設定により記録可能
10.2.2	ルート権限または管理権限を持つ個人によって行われたすべてのアクション	・DBに対する DB 管理者によるすべてのアクセスを記録する。	○	• log_statement 設定により記録可能
10.2.3	すべての監査証跡へのアクセス	・DB 監査証跡に対するすべてのアクセスを記録する。	△	• 監査証跡相当のログデータに対するアクセス記録は OS 機能を使って実現する 以下の項にて詳細を解説: [4.6 出力したログの保全(改ざん防止)]
10.2.4	無効な論理アクセス試行	・DBに対する以下のような無効な論理アクセスを記録する。 例) ・SQL 構文エラー ・存在しないオブジェクトに対するアクセス ・権限のないオブジェクトに対するアクセス	○	• log_min_messages 設定により左記のようなエラーをログ記録できる
10.2.5	識別および認証メカニズムの使用	・DB に対するログイン／ログアウトを記録する。	○	• log_connection、log_disconnection 設定で記録できる
10.2.6	監査ログの初期化	・DB 監査証跡の初期化、設定変更などの操作を記録する。	△	• ログ取得設定変更の記録は OS 機能を使って実現する 以下の項にて詳細を解説: [4.6 出力したログの保全(改ざん防止)]
10.2.7	システムレベルオブジェクトの作成および削除	・DB オブジェクト(DB アカウント、テーブル、ビュー など)の作成、変更などの操作 を記録する。	○	• log_statement 設定により記録できる
10.3	イベントごとに、すべてのシステムコンポーネントについて少なくとも以下の監査証跡エントリを記録する。		−	• 10.3 以下の諸項目は PostgreSQL 標準機能による対応のほか、監査ログ拡張機能を提供する拡張モジュールや商用ソフトウェアも利用可能

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
				以下の項にて詳細を解説: [4.2 pgaudit 拡張モジュールの使用] [4.3 オブジェクト監査] [4.4 PostgreSQL を拡張した商用製品による監査]
10.3.1	ユーザ識別	・DB アカウント名(ID)、OS アカウント名、アプリケーションアカウント名など	△	• log_line_prefix 設定により記録できる、ただし OS アカウント記録機能がない
10.3.2	イベントの種類	・SQL 文、SQL タイプ	○	• log_statement 設定により記録できる
10.3.3	日付と時刻	・操作日時	○	• log_line_prefix 設定により記録できる
10.3.4	成功または失敗を示す情報	・操作結果、エラーコード	○	• log_line_prefix 設定により記録できる
10.3.5	イベントの発生元	・DB クライアントの IP アドレス、マシン名など	○	• log_line_prefix 設定により記録できる
10.3.6	影響を受けるデータ、システムコンポーネント、またはリソースの ID または名前	・オブジェクト名、オブジェクト ID、カラム名、カラム ID など	×	• 標準機能では記録できない • SQL 文の記録をもって代替する
10.5	変更できないよう、監査証跡をセキュリティで保護する。		−	−
10.5.1	監査証跡の表示を、仕事関連のニーズを持つ人物のみに制限する	・DB 監査証跡にアクセスできる人物を最小限に制限する。	○	• そのように運用可能
10.5.2	監査証跡ファイルを不正な変更から保護する。	・DB 監査証跡にアクセスできる人物を最小限に制限する。 ・DB 監査証跡の改ざん対策を講じる。 例) ・ログの多重化 ・電子証明書(タイムスタンプなど)の付加	△	• 監査証跡記録の不正な変更に対する保護は OS 機能を使って実現する 以下の項にて詳細を解説: [4.5 DBMS 一般情報へのアクセス情報の取得] [4.6 出力したログの保全(改ざん防止)]
10.5.3	監査証跡ファイルを、変更が困難な一元管理ログサーバまたは媒体に即座にバックアップする。	・DB 監査証跡を左記の運用で管理する。 例) ・書き換え不能ストレージの使用	○	• そのように運用可能
10.5.5	ログに対してファイル整合性監視または変更検出ソフトウェアを使用して、既存のログデータを変更すると警告が生成されるようにする(ただし、新しいデータの追加は警告を発生させない)。	・DB 監査証跡の改ざん対策を講じる。	△	• 監査証跡記録の不正な変更に対する保護は OS 機能を使って実現する 以下の項にて詳細を解説: [4.6 出力したログの保全(改ざん防止)]
10.6	少なくとも日に一度、すべてのシステムコンポーネントのログを確認する。ログの確認には、侵入検知システム	・DB においても、ログ解析・検知ツールを使用して、少なくとも日に一度、DB 監査証跡の内容を確認し不審な操作を検出する。	○	• そのように運用可能

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
	(IDS)や認証、認可、アカウンティングプロトコル(AAA)サーバ(RADIUS など)のようなセキュリティ機能を実行するサーバを含める必要がある。 注: 要件 10.6 に準拠するために、ログの収集、解析、および警告ツールを使用することができます。			
10.7	監査証跡の履歴を少なくとも 1 年間保持する。少なくとも 3 カ月はすぐに分析できる状態にしておく(オンライン、アーカイブ、バックアップから復元可能など)。	・DB 監査証跡を少なくとも 1 年間保持する。 ・少なくとも 3 ヶ月は、すぐに DB 監査証跡を分析できる状態にしておく。	○	• そのように運用可能
要件:11	セキュリティシステムおよびプロセスを定期的にテストする。		―	―
11.2	内部および外部ネットワークの脆弱性スキャンを少なくとも四半期に一度およびネットワークでの大幅な変更(新しいシステムコンポーネントのインストール、ネットワークトポロジの変更、ファイアウォール規則の変更、製品アップグレードなど)後に実行する。 注: 四半期に一度の外部の脆弱性スキャンは、PCI(Payment Card Industry)セキュリティ基準審議会(PCI SSC)によって資格を与えられた ApprovedScanningVendor(ASV)によって実行される必要があります。ネットワーク変更後に実施されるスキャンは、会社の内部スタッフによって実行することができます。	・DB サーバ(OS レベル、DBMS 製品)、及びデータが格納されるインスタンスに対して、少なくとも四半期に一度脆弱性をスキャンする。	○	• そのように運用可能
11.3	外部および内部のペネトレーションテストを少なくとも年に一度および大幅なインフラストラクチャまたはアプリケーションのアップグレードや変更(オペレーティングシステムのアップグレード、環境へのサブネットワー		―	―

#	PCI DSS 要件	DBSC 対応要件	PostgreSQL 対応	
			対応レベル	対応策
	クの追加、環境への Web サーバの追加など）後に実行する。これらのペネトレーションテストには以下を含める必要がある。			
11.3.2	アプリケーション層のペネトレーションテスト	・DB サーバ(OS レベル、DBMS 製品)、及びデータが格納されるインスタンスに対して、少なくとも年に一度ペネトレーションテストを実施する。	○	• そのように運用可能
要件：12	従業員および派遣社員向けの情報セキュリティポリシーを整備する。		－	－
12.1	以下を実現するセキュリティポリシーを確立、公開、維持、および周知する。		－	－
12.1.2	脅威、脆弱性、結果を識別する年に一度のプロセスを正式なリスク評価に含める。	・DB セキュリティ対策の有効性を評価する。	－	－
12.1.3	レビューを少なくとも年に一度含め、環境の変化に合わせて更新する。	・環境の変化に合わせて DB アカウントを見直す。 例） ・不適切なアカウントや権限を再度整理する。	○	• そのように運用可能
12.2	この仕様の要件と整合する日常的な運用上のセキュリティ手順を作成する（たとえば、ユーザアカウント保守手順、ログレビュー手順）。	・DB 管理業務のチェック。 例） ・管理業務の記録を残す。 ・管理者のローティーションを実施する。	○	• そのように運用可能
12.5	個人またはチームに以下の情報セキュリティ管理責任を割り当てる。		－	－
12.5.4	追加、削除、変更を含め、ユーザアカウントを管理する	・DB アカウント管理する。 例） ・DB 利用者の整理。 ・DB アカウントの整理。	○	• そのように運用可能
12.5.5	データへのすべてのアクセスを監視および管理する。	・DB アクセスログを監視および管理する。 例） ・ログの取得の目的。 ・ログ取得対象アクセスの整理。	○	• そのように運用可能

4. 調査結果詳細

本章では、前章「3.PCI DSS への PostgreSQL 対応調査結果一覧」の中で、要件の機能は PostgreSQL に標準で備わっていないが、何らかの代替対応策を講じて対応可能であるとした項目について、代替対応策の具体的な方法を示します。本章の各節は独立しています。前章の「表 3.2: PCIDSS 要件への PostgreSQL 対応」の各項目で関連する本章の節が示しています。また、逆に本章各節の冒頭で「対応する PCI DSS 要件」として、PCI DSS 要件の項目番号を示しています。

本章各節では、以下のように囲みでコマンド操作と出力、または、ファイル内容を記載します。コマンド操作例の行頭の $ または # は、一般ユーザまたは root ユーザのプロンプトをあらわします。

コマンド操作と出力の例:

```
$ pg_ctl start -w
waiting for server to start....
   ～中略～
server started
```

ファイル内容例(mapfile):

```
# MAPNAME     SYSTEM-USERNAME     PG-USERNAME
map1          /user[1-2]          user1          #設定①
map1          user3               user2          #設定②
```

本書では、PostgreSQL 9.3 メジャーバージョンを対象とし、また、Redhat Enterprise Linux 6.5 を想定環境としています。本章の記述も特記無い限り、これらバージョン・環境を前提とした記載となります。

4.1. アカウントポリシー機能の実現

対応する PCI DSS 要件	8.5.3、8.5.9、8.5.10、8.5.11、8.5.12、8.5.13、8.5.14

PostgreSQL 本体機能ではアカウントポリシー機能が十分に備わっておらず、以下を実現することができません。

表 4.1: PostgreSQL に備わっていないアカウントポリシー

最初にパスワード変更を行うことを強制する
少なくとも 90 日ごとにユーザパスワードを変更する。
パスワードに 7 文字以上が含まれることを要求する。
数字と英文字の両方を含むパスワードを使用する。
ユーザが新しいパスワードを送信する際、最後に使用した 4 つのパスワードと同じものを使用できないようにする。
最大 6 回の試行後にユーザ ID をロックアウトして、アクセス試行の繰り返しを制限する。
ロックアウトの期間を、最小 30 分または管理者がユーザ ID を有効にするまで、に設定する

しかしながら、PostgreSQL のログイン認証に外部の認証サーバ（ディレクトリサーバ）を使う設定をすることで、これら機能を実現できます。PostgreSQL では LDAP、GSSAPI、RADIUS といったプロトコルにて外部認証サーバを利用可能です。各種認証サーバソフトウェアはオープンソースの実装を含め、アカウントポリシー機能を豊富に備えています。

ここでは最もシンプルな LDAP を使う設定例を紹介します。LDAP サーバのオープンソース実装である OpenLDAP を動作させて、実際に動かす例を示します。

4.1.1. 構成例の概要

下図のように PostgreSQL が認証に OpenLDAP を使用する構成を作成します。

LDAP ディレクトリ内の組織単位「ou=dbusers,dc=example,dc=com」以下に PostgreSQL 用ユーザアカウントのエントリーを配置してあるものとします。また、LDAP サーバは PostgreSQL と同ホストで稼働しているものとします。

図 4.1: OpenLDAP との連携

4.1.2. PostgreSQL の設定

PostgreSQL が configure にて --with-ldap オプションを指定してビルドされている必要があります。また、--with-ldap オプションを付けてビルドするには Unix、Linux では OpenLDAP がインストールされている必要があります。ただし、ここで OpenLDAP を使うのは LDAP のクライアント側ライブラリを利用するためであって、LDAP サーバ側は別のソフトウェアであって構いません。なお、PostgreSQL 公式リポジトリや Redhat Enterprise Linux の RPM パッケージでは、--with-ldap オプションが付加されてビルドされています。

PostgreSQL の接続で LDAP 認証を使用するには pg_hba.conf の認証メソッドとオプションに以下のように記述

します。

```
# TYPE   DATABASE   USER   ADDRESS    METHOD  OPTIONS
  host   all        all    0.0.0.0/0  ldap    ldapserver=127.0.0.1 ldapprefix="cn="
ldapsuffix=",ou=dbusers,dc=example,dc=com" ldaptls=0
                                                          (上記は実際には 1 行です)
```

これで PostgreSQL の user1 ユーザに対する接続時認証で、LDAP サーバに対して「cn=user1,ou=dbusers,dc=example,dc=com」というエントリに対するパスワード認証が行われるようになります。パスワードは LDAP サーバ側で保持されているエントリ毎のパスワードが使われます。この書き方は単純バインドと呼ばれます。

外部ホストの LDAP サーバを使うのであれば「ldaptls=1」を指定して、TLS による暗号化通信を使うのが良いでしょう。この場合、LDAP サーバ側でも TLS を有効にしておく必要があります。

LDAP 認証には他にもオプションがいくつかあります。詳しくは PostgreSQL マニュアル [19.3.8. LDAP 認証] [13] を参照してください。

4.1.3. OpenLDAP の設定

PostgreSQL の外部認証を行うための OpenLDAP 構築手順を示します。あくまで PostgreSQL 認証の動作確認用ですので一般的な OpenLDAP の導入・運用について詳しく知りたい場合には、別の文献を参照してください。

以下のパッケージが導入済みであるものとします。

- openldap-2.4.39
- openldap-clients-2.4.39
- openldap-devel-2.4.39
- openldap-servers-2.4.39

以下二つのファイルを編集して管理者の仮パスワードとドメインを記述します。

- /etc/openldap/slapd.d/cn=config/olcDatabase={0}config.ldif

```
olcRootPW: secret                              (左記の行を追加)
```

- /etc/openldap/slapd.d/cn=config/olcDatabase={2}bdb.ldif

```
olcSuffix: dc=example,dc=com                   (olcSuffix: の項目を書換え)
olcRootDN: cn=Manager,dc=example,dc=com        (olcRootDW: の項目を書換え)
olcRootPW: secret                              (左記の行を追加)
```

以下コマンドで LDAP サーバを起動します。警告が出ているのは、設定ファイルを直接編集しているためです。起動して以降は LDAP 操作を通して設定変更していきます。そうしますと警告は出なくなります。

```
# service slapd start
slapd の設定ファイルをチェック中:                          [警告]
548fe558 ldif_read_file: checksum error on
"/etc/openldap/slapd.d/cn=config/olcDatabase={0}config.ldif"
548fe558 ldif_read_file: checksum error on
"/etc/openldap/slapd.d/cn=config/olcDatabase={2}bdb.ldif"
config file testing succeeded
slapd を起動中:                                            [  OK  ]
```

以下、ldapmodify コマンドで管理者パスワードを変更する手順を例示します。

```
# slappasswd
New password:                                          (設定したいパスワードを入力)
Re-enter new password:
{SSHA}glZ4S3i8bgk6IjD7y+Mv2qdp4X/oRW0m

# cat > config_pw_mod.ldif <<'EOS'
```

13 http://www.postgresql.jp/document/9.3/html/auth-methods.html#AUTH-LDAP

```
dn: olcDatabase={0}config,cn=config
changetype: modify
replace: olcRootPW
olcRootPW: {SSHA}glZ4S3i8bgk6IjD7y+Mv2qdp4X/oRW0m
EOS

# cat > config_pw_mod_bdb.ldif <<'EOS'
dn: olcDatabase={2}bdb,cn=config
changetype: modify
replace: olcRootPW
olcRootPW: {SSHA}glZ4S3i8bgk6IjD7y+Mv2qdp4X/oRW0m
EOS

# ldapmodify -x -W -D 'cn=config' -f config_pw_mod.ldif
Enter LDAP Password:                                    (仮パスワード secret を入力)
modifying entry "olcDatabase={0}config,cn=config"

# ldapmodify -x -W -D 'cn=config' -f config_pw_mod_bdb.ldif
Enter LDAP Password:                    (既に変更されているので変更後パスワードを入力)
modifying entry "olcDatabase={0}config,cn=config"
```

続いてサンプルデータを投入します。ドメインと管理ユーザ、データベースユーザ用の組織単位を登録し、その中にエントリ「cn=user1,ou=dbusers,dc=example,dc=com」と「cn=user2,ou=dbusers,dc=example,dc=com」を登録しています。

```
# cat > data.ldif <<'EOS'
dn: dc=example,dc=com
objectClass: dcObject
objectClass: organization
o: example.com
dc: example

dn: cn=Manager,dc=example,dc=com
objectclass: organizationalRole
cn: Manager

dn: ou=dbusers,dc=example,dc=com
ou: dbusers
objectClass: organizationalUnit

dn: cn=user1,ou=dbusers,dc=example,dc=com
objectclass: person
sn: User1
cn: user1

dn: cn=user2,ou=dbusers,dc=example,dc=com
objectclass: person
sn: User2
cn: user2
EOS

# ldapadd -x -W -D 'cn=Manager,dc=example,dc=com' -f data.ldif
Enter LDAP Password:
adding new entry "dc=example,dc=com"
adding new entry "cn=Manager,dc=example,dc=com"
adding new entry "ou=dbusers,dc=example,dc=com"
adding new entry "cn=user1,ou=dbusers,dc=example,dc=com"
adding new entry "cn=user2,ou=dbusers,dc=example,dc=com"
```

ldappasswd コマンドでパスワードを設定します。以下例では「passuser1」「passuser2」というパスワードを設定しています。

```
# ldappasswd -x -W -D 'cn=Manager,dc=example,dc=com' -s 'passuser1' ¥
  'cn=user1,ou=dbusers,dc=example,dc=com'

# ldappasswd -x -W -D 'cn=Manager,dc=example,dc=com' -s 'passuser2' ¥
  'cn=user2,ou=dbusers,dc=example,dc=com'
```

これで、以下のように user1、user2 ユーザで接続したとき、上記で LDAP にて設定したパスワードで認証されるようになります。

```
# su - postgres
$ createuser user1
$ createuser user2
$ psql -h 127.0.0.1 -U user1 -d postgres
Password for user user1:
```

さらに各ユーザで自分のパスワード変更ができるようにアクセス権限を設定します。これにより、各ユーザで自身のパスワードを変更できます。

```
# cat <<EOS > config_access.ldif
dn: olcDatabase={2}bdb,cn=config
changetype: modify
replace: olcAccess
olcAccess: to attr=userPassword by self write by anonymous auth by * none
olcAccess: to * by self write by users read by anonymous auth
EOS

# ldapmodify -x -W -D 'cn=config' -f config_access.ldif
Enter LDAP Password:
```

```
$ ldappasswd -x -W -D 'cn=user1,ou=dbusers,dc=example,dc=com' -S ¥
   'cn=user1,ou=dbusers,dc=example,dc=com'
New password:
Re-enter new password:
Enter LDAP Password:
```

4.1.4. アカウントポリシーの設定

OpenLDAP にはアカウントポリシー（OpenLDAP ドキュメントではパスワードポリシーという言い方をしています）を実現する ppolocy というモジュールが用意されています。これを導入します。

以下コマンドでは、モジュールを追加する設定、デフォルトのポリシーとして使う設定、ポリシー用組織単位とポリシー定義の設定をしています。

```
# cat > config_add_ppol_module.ldif <<'EOS'
dn: cn=module{0},cn=config
objectClass: olcModuleList
cn: module{0}
olcModuleLoad: ppolicy.la
EOS

# ldapadd -x -W -D 'cn=config' -f config_add_ppol_module.ldif


# cat > config_add_ppol.ldif <<'EOS'
dn: olcOverlay=ppolicy,olcDatabase={2}bdb,cn=config
objectClass: olcPPolicyConfig
olcOverlay: ppolicy
olcPPolicyDefault: cn=default,ou=policies,dc=example,dc=com
olcPPolicyUseLockout: TRUE
EOS

# ldapadd -x -W -D 'cn=config' -f config_add_ppolicy.ldif


# cat > policies_ou.ldif <<'EOS'
dn: ou=policies,dc=example,dc=com
objectClass: organizationalUnit
objectClass: top
ou: policies
EOS

# ldapadd -x -W -D 'cn=Manager,dc=example,dc=com' -f policies_ou.ldif
```

```
# cat > policies.ldif <<'EOS'
dn: cn=default,ou=policies,dc=example,dc=com
objectClass: top
objectClass: device
objectClass: pwdPolicy
cn: default
pwdAttribute: userPassword
pwdMaxFailure: 4
pwdMustChange: FALSE
pwdMinLength: 6
pwdInHistory: 1
pwdCheckQuality: 1
pwdMinAge: 0
pwdLockout: TRUE
pwdMaxAge: 2592000
EOS

# ldapadd -x -W -D 'cn=Manager,dc=example,dc=com' -f policies.ldif
```

この設定では「pwdMaxFailure: 4」となっています。パスワードを４回続けて間違えると、そこでアカウントロックされるというものです。これで動作確認してみます。

```
$ psql -U user2 -h 127.0.0.1 -d postgres
Password for user user2:  （わざとパスワードを間違える）
psql: FATAL:  LDAP authentication failed for user "user2"
```

上記を4回繰り返すと、5回目に正しくパスワードを入れても認証エラーになります。OpenLDAP 側で以下のように管理者からパスワードを設定しなおすと、再度接続可能になります。

```
# ldappasswd -x -W -D 'cn=Manager,dc=example,dc=com' -s newpass2 ¥
   'cn=user2,ou=dbusers,dc=example,dc=com'
```

この他、OpenLDAP の ppolocy モジュールでは次のようなことができます。

- 最初にユーザがパスワード変更することを強制
- 繰り返しパスワードを誤ったときのアカウントロックと、指定時間経過後の自動解除の設定
- パスワードの有効期間を設定する
- 最後に設定したN種類のパスワードと同じパスワードに設定することを禁止する
- パスワードの長さが指定より長いことを必須とする
- 「数字、英文字の両方を含むこと」など、パスワード品質を満たすかチェックする

ppolicy モジュールが提供するアカウントポリシー機能の詳細は、OpenLDAP Software Administrator's Guide [12.10. Password Policies] [14]参照してください。

14 http://www.openldap.org/doc/admin24/overlays.html#Password%20Policies

4.2. pgaudit 拡張モジュールの使用

対応する PCI DSS 要件	10.2, 10.3

PostgreSQL では log_statement パラメータにより監査証跡を出力する事ができますが、指定方法が限られています。例えば、参照の SQL 文(SELECT / COPY TO)のみを対象とする事ができません。
pgaudit 拡張モジュールを導入する事により、きめ細かな指定ができます。例えば、参照の SQL 文(SELECT / COPY TO)や、アクセス権限付与に関する SQL 文 (GRANT / REVOKE)のみを対象とする事ができます。
PostgreSQL 本体とは別に github からダウンロードしてインストールし、PostgreSQL のエクステンションとして実装されます。

なお 4.2.4 では 2016 年 3 月現在、コミュニティで開発が進んでいる pgaudit についても紹介します。

4.2.1. pgaudit 導入および設定

pgaudit の導入手順を示します。

1. pgaudit のソースダウンロードおよび PostgreSQL サーバへのアップロード
2ndQuadrant の GitHub サイト[15]から pgaudit の zip ファイルををダウンロードします。
(確認例)

```
$ ls -l pgaudit*.zip
-rw-r--r-- 1 root root 17269  3月  5 22:16 pgaudit-master.zip
```

2. pg_audit のコンパイル
zip ファイルを解凍しコンパイルします。
(実行例)　コマンドのみ列記します。

```
$ unzip pgaudit-master.zip
$ cd pgaudit-master
$ make USE_PGXS=1
$ make USE_PGXS=1 install
```

エクステンション作成に必要なファイルが $PGHOME/share/postgresql/extension 配下に作成されます。
（環境によりディレクトリ構成は異なる場合があります）。

3. 共有ライブラリの設定（パラメータ）
postgresql.conf にて shared_preload_libraries パラメータに pgaudit を設定し、起動します。
(設定例)

```
shared_preload_libraries = 'pgaudit'
```

(実行例)

```
$ pg_ctl start -w
waiting for server to start....
   ～中略～
server started
```

4. pgaudit エクステンションの作成
psql で pgaudit エクステンションを作成するデータベースに接続します。エクステンションを作成するユーザには SUPERUSER 権限が付与されている必要があります。

(実行例)　superuser1 ユーザには SUPERUSER 権限が付与されています。

```
$ psql -U superuser1 database1

=# CREATE EXTENSION pgaudit;
CREATE EXTENSION
```

15 https://github.com/2ndQuadrant/pgaudit

pgaudit のライブラリがロードされていない場合は以下のエラーが発生します。

(実行例)

```
=# CREATE EXTENSION pgaudit;
ERROR:  pgaudit must be loaded via shared_preload_libraries
```

5. 監査SQLのカテゴリを設定（パラメータ設定）

監査SQLのカテゴリを pgaudit.log パラメータに設定します。カンマ区切りで複数のカテゴリを指定できます。

(設定例)

```
pgaudit.log = 'read, write, privilege, user, definition, config, admin, function'
```

リロードで反映します。

(実行例)

```
$ pg_ctl reload
server signaled
```

表 4.2: pgaudit における SQL のカテゴリ

カテゴリ	説明 (例)
read	データベースオブジェクトを参照する SQL 文。監査証跡におけるクラスは”READ”。 (例: SELECT / COPY TO) “SELECT now() ;” はデータベースオブジェクトを参照しないため対象外となる。
write	データベースオブジェクトを更新する SQL 文(DML)。監査証跡におけるクラスは”WRITE”。 (例: SELECT / INSERT / UPDATE / COPY FROM / TRUNCATE)
privilege	アクセス権限に関する DDL 文。監査証跡におけるクラスは”PRIVILEGE”。 (例: GRANT / REVOKE)
user	データベースユーザに関する DDL 文。監査証跡におけるクラスは”USER”。 (例: CREATE / DROP / ALTER ROLE)
definition	ユーザレベルの DDL 文 。監査証跡におけるクラスは”DEFINITION”。 (例: CREATE / ALTER / DRO0P TABLE)
config	データベースの設定に影響する管理者レベルの SQL 文 。監査証跡におけるクラスは”CONFIG”。 (例: CREATE LANGUAGE / CREATE OPERATOR)
admin	データベースの設定に影響しない管理者レベルの SQL 文。監査証跡におけるクラスは”ADMIN”。 (例: CLUSTER / VACUUM / REINDEX)
function	関数が実行された場合。監査証跡におけるクラスは”FUNCTION”。 関数内で実行された SQL 文も監査証跡として出力される。 (例 SELECT update_filler(1,'xxxxx');) 上記の update_filler 関数内にて、SELECT 文 1 回および UPDATE 文 1 回を実行した場合、 監査証跡としては、FUNCTION,READ,WRITE の 3 レコードが出力される。

4.2.2. pgaudit の動作確認

監査の動作確認として update_filler 関数を実行します。この関数内では、以下の処理を行います。

pgbench_accounts テーブルに対して SELECT （1 回）

pgbench_accounts テーブルに対して UPDATE（1 回）

(関数例)

```
CREATE OR REPLACE FUNCTION update_filler(p_aid int,p_filler TEXT)
RETURNS VOID AS $$
DECLARE
  v_filler text;
BEGIN
  SELECT filler INTO v_filler FROM pgbench_accounts WHERE aid = $1;      --- READ
  UPDATE public.pgbench_accounts SET filler = $2 WHERE aid = $1;         --- WRITE
END;
$$  LANGUAGE plpgsql;
```

(実行例)

```
=# SELECT update_filler(1,'xxxxx');
 update_filler
---------------

```

サーバログには以下の出力があります。

(出力例)便宜上、間に空白行を挿入

```
LOG:  AUDIT,2015-02-11 20:03:45.461146+09,database1,user1,user1,FUNCTION,EXECUTE,
FUNCTION,public.update_filler,SELECT update_filler(1,'xxxxx');   --- ①

LOG:  AUDIT,2015-02-11 20:03:45.462158+09,database1,user1,user1,READ,SELECT,TABLE,
public.pgbench_accounts,SELECT update_filler(1,'xxxxx');         --- ②

LOG:  AUDIT,2015-02-11 20:03:45.462848+09,database1,user1,user1,WRITE,UPDATE,TABLE,
public.pgbench_accounts,SELECT update_filler(1,'xxxxx');         --- ③
```

“LOG: AUDIT,”<timestamp>,<database>,<username>,<effective username>,<class>,<tag>,
<object type>,<object id>,<command text>

表 4.3: pgaudit の出力項目　（③の場合）

項目	説明
timestamp	SQL が開始された時間。 (③ の場合 '2015-02-11 20:03:45')
database	SQL が実行されたデータベース。(③ の場合 database1)
username	SQL を実行したユーザ。(③ の場合 uesr1)
effective username	SQL を実行した実ユーザ名。通常は SQL を実行したユーザと同一ですが、SET ROLE 文が実行された場合に差異が発生します。(③ の場合 user1)
class	pgaudit における SQL の分類。 (例 READ / WRITE / PRIVILEGE / USER / DEFINITION / CONFIG / ADMIN / FUNCTION) (③ の場合 WRITE)
tag	SQL のタグ。(例 SELECT / INSERT / UPDATE / DEELTE / EXECUTE など) (③ の場合 UPDATE)
object type	処理対象のオブジェクトのタイプ。例) TABLE / VIEW / FUNCTION など (③ の場合 TABLE)
object id	スキーマ付のオブジェクト名。(③ の場合 public.pgbench_accounts)
commnad text	SQL 文。関数から実行された場合は関数の実行文。 (上記③の場合 SELECT update_filler(1,'xxxxx');)

4.2.3. pgaudit の注意点

- SQL 文が発行された時点でサーバログに出力するため、以下の場合も出力される事にご注意下さい。
  - SQL がエラーで終了
  - SQL実行中にキャンセルされた
  - トランザクションがロールバックされた

4.2.4. コミュニティでの pgaudit の開発動向

ここでは、2016 年 3 月現在、コミュニティで開発が進んでいる pgaudit の概要を紹介します。
4.2.3 までで説明してきた pgaudit を元にして、2014 年の初めごろから、コミュニティでは PostgreSQL に監査機能を実現する contrib モジュールの一つとして、”pg_audit”が提案されて、議論されてきました。議論の結果、contrib モジュールへの取り込みは見送られ、2015 年 7 月から”pgaudit”という名前で開発が続いています[16]。
現在の開発は、https://github.com/pgaudit/pgaudit で続いています。こちらの pgaudit を本報告書では便宜上 “pgaudit2”と呼ぶことにします。

pgaudit2 の特長について
コミュニティに”pg_audit”が提案されていた時点で、細かい粒度でログ出力を制御するために、Object Auditing mode が追加されました。それが pgaudit2 にも引き継がれています。これに対して、従来からあるログの取得方式を”Session Auditing mode”と呼んでいます。
Object Auditing mode は、次のような特徴を持っています。

1. 特定のテーブルやカラムを操作した SQL 文に対するログだけを取得することができる
   一般に監査のために、データベースに投入された全ての SQL 文をログとして取得すると、ログデータ量が多大となり、運用上および、事後のログの監査の際に多大な負担となる課題に対処するための機能です。
2. 対象となる操作は、SELECT, INSERT, UPDATE, DELETE の 4 種に限られる
   Object auditing mode の設定では、データベースの権限機能に似た考え方を使って、ログの対象となる表を絞り込みます。具体的には、postgresql.conf にて以下のように指定されたロールに対して、以下のように監査対象のテーブルに対する権限を付与します。[17]

（postgresql.conf 設定例）

```
pgaudit.role = 'auditor'
```

リロードで反映します。
(実行例）

```
$ pg_ctl reload
server signaled
```

(権限付与例）

```
=# GRANT select, delete ON public.account TO auditor;
GRANT
```

このように設定することで、対象テーブル”public.account”に対する SELECT および DELETE 操作のみの監査を取得することができます。

このような機能を持った pgaudit2 は、現在も開発が続いており、今後の展開が期待されるツールとなっています。

16 大山 真実, “監査要件を有するシステムに対する PostgreSQL 導入の課題と可能性”, PostgreSQL カンファレンス 2015 発表資料, 27. Nov. 2015, 東京, http://www.postgresql.jp/events/jpug-pgcon2015/detail#M3
17 Abhijit Menon-Sen, Ian Barwick, and David Steele, “PostgreSQL Audit Extension”, https://github.com/pgaudit/pgaudit/blob/master/README.md

4.3. オブジェクト監査

対応する PCI DSS 要件	10.2.1

PostgreSQL 本体機能だけを用いる場合、PCI DSS の 10.2.1 に掲げる以下の要件を十分効率的に実施することができません。

カード会員データへの全ての個人アクセスを追跡および監視する

PostgreSQL 本体機能だけを用いる場合、特定の表に対する操作に限ってログを取得することができないため、表 3.2 に示すように、log_statement を all などに指定する事になります。この時は、「カード会員個人データ」のような特定に格納されている情報に対するアクセス以外の、全ての表に対するアクセスのログが出力されます。出力されるログデータの総量は、監査すべきデータに対して極めて大きくなり、監査ログの保全、確認の作業に余分のコストを費やす必要があります。

このようなケースでは、一般に「オブジェクト監査」や「ファイングレイン監査[18]」と呼ばれる、データベース上の特定のオブジェクト(表)に対するアクセスのみを監査ログとして取得する機能が望まれます。ここでは、先に 4.2.4 節で紹介した pgaudit を用いたオブジェクト監査を紹介します。

pgaudit には、”object auditing”という機能を備えています[17]。この機能を用いることで、事前に指定した特定の表に対する SELECT, INSERT, UPDATE, DELETE がなされた場合に、その操作ログを出力することができます。また、実表だけでなく VIEW に対する操作も同様にログを取得することができるため、利用者に応じた VIEW を設定して重要情報へのアクセスを制限している場合にも、適切なログを取得することができます。ただし、表そのものを削除する TRUNCATE についてはログを出力しないため、この操作のログ取得については別の対策を講じる必要があります。

pgaudit によるオブジェクト監査のための設定方法等は 4.2.4 や pgaudit のマニュアル[20]をご覧ください。

18 Oracle データベースの提供する機能。『Oracle データベースセキュリティガイド』(http://docs.oracle.com/cd/E16338_01/network.112/b56285/auditing.htm)などを参照。

4.4. PostgreSQL を拡張した商用製品による監査

対応する PCI DSS 要件	10.2, 10.3

PostgreSQL 本体は監査情報も他の情報も同一のサーバログに出力される仕様であり、専用の監査ログに出力する機能はありません。専用の監査ログであれば、監査データの検索や監査ログの保存を行い易くなります。
対処策の 1 つとして PostgreSQL を拡張して監査専用のログ出力機能を付加した商用製品を使う方法があります。本項目では、監査ログ出力機能を備えた EDB Postgres（旧 Postgres Plus)製品について紹介します。

4.4.1. EDB Postgres

EDB Postgres は米国 EnterpriseDB 社[19]（以下 EDB 社）の製品です。PostgreSQL をベースとしており、Oracle Database との互換性の高さと豊富な GUI ツールが特徴の商用のデータベースです。
EDB 社には PostgreSQL のコミッターが多数在籍しており、PostgreSQL コミュニティにも貢献しています。

4.4.2. EDB Postgres における監査ログ出力機能

EDB Postgres 特有の主な監査ログの機能を紹介します。

- サーバログとは別の出力先を設定できます。
  パラメータ edb_audit_directory および edb_audit_filename にて任意の出力先を指定できます。

- 監査対象をより柔軟に指定できます。
  PostgreSQL のパラメータ log_statement {none | ddl | mod | all} による指定では柔軟性が十分でなく、例えば SELECT 文のみを指定する事ができません。
  Postgres Plus ではパラメータ edb_audit_statement {none | ddl | dml | select | error | all} にて SELECT 文のみの指定ができます。

オブジェクトを指定した監査（オブジェクト監査）の機能は、EDB Postgres には実装されていません。

19 http://www.enterprisedb.co.jp

4.5. DBMS 一般情報へのアクセス情報の取得

対応する PCI DSS 要件	10.2.6

PostgreSQL 本体の設定ファイルへのアクセスログを取得する機能は実装されておりません。
PostgreSQL の設定ファイルへのアクセスログを残したい場合、audit の機能を用いて実現できます。audit は daemon プロセスとして起動して対象ファイルの監視をします。audit の詳細は以下のページをご参照ください。
https://docs.oracle.com/cd/E39368_01/b72804/ol_audit_sec.html
以下の手順は audit のインストールと PostgreSQL の設定ファイルのアクセス監視をする設定を記載したものです。

1. audit がインストールされているか確認します。

   audit がインストールされていない場合は 2.の手順でインストールします。

```
# rpm -qa | grep audit                                パッケージが入っているか確認します。
```

2. audit をインストールします。

   yum コマンドでインストールします。
   クローズな NW の場合、rpm ファイルを取得して rpm -i コマンドでインストールしてください。

```
# sudo yum install audit
Loaded plugins: fastestmirror, versionlock
Setting up Install Process
Loading mirror speeds from cached hostfile
epel/metalink
| 6.2 kB     00:00
 * base: centos.mirror.secureax.com
 * epel: epel.mirror.srv.co.ge
 * extras: ftp.iij.ad.jp
 * updates: ftp.iij.ad.jp
省略
Installed:
  audit.x86_64 0:2.3.7-5.el6

Complete!


# rpm -qa audit                                       インストールできたことを確認します。
audit-2.3.7-5.el6.x86_64
```

3. auditの設定にアクセス監視の設定をします。

   設定例) PostgreSQLの設定ファイルへのアクセス監視ルール

   audit.rulesにpostgresql.confのアクセス監視ルールを追加します。
   今回の設定はPostgreSQLの設定ファイルへのアクセスの証跡が残るように設定します。どのユーザでも
   postgresql.confへのアクセスが成功した場合にログが出力されます。
   ${PGDATA}はPostgreSQLのデータベースクラスタのディレクトリパスです。

```
# sudo sh -c "echo '-a exit,always -F arch=b64 -S open -F path=${PGDATA}/postgresql.conf -F
success=1' >> /etc/audit/audit.rules"


# sudo cat /etc/audit/audit.rules                                    ルールが追加されたことを確認します。
省略
# Feel free to add below this line. See auditctl man page
-a exit,always -F arch=b64 -S open -F path=/dbfp/pgdatadata/postgresql.conf -F success=1
```

4. auditの設定を反映します。

```
--- auditdのプロセスが起動しているか確認します。
# ps aux | grep auditd
root       596  0.0  0.0      0     0 ?        S    10:23   0:00 [kauditd]
root      3207  0.0  0.1  27596   804 ?        S<sl 15:03   0:00 auditd
pgsql    3225  0.0  0.1 103248   864 pts/0    S+   15:04   0:00 grep auditd

--- プロセスが起動している場合はrestartを発行します。
# sudo service auditd restart
Stopping auditd:                                           [  OK  ]
Starting auditd:                                           [  OK  ]

--- プロセスが起動していない場合はstartを発行します。
# sudo service auditd start
Starting auditd:                                           [  OK  ]
```

5. 検出例) vi エディターで postgresql.conf を編集したときのアクセスログ

   アクセスログの参照観点は、name が編集したファイルパス、comm が編集したアプリケーション、acct が編集したユーザ、msg が変種時間を表します。編集した時間は UNIX 時間で出力されます。UNIX 時間については以下の URL を参照してください。また、UNIX 時間の変換サービスもあるのでご参照ください。

   UNIX 時間: *http://ja.wikipedia.org/wiki/UNIX%E6%99%82%E9%96%93*

   UNIX 時間変換ツール: http://www.math.kobe-u.ac.jp/~kodama/tips-DateTime-transform.html

```
# vim でアクセスします。
# vim ${PGDATA}/postgresql.conf

# 出力されているログを確認します。
# sudo tail /var/log/audit/audit.log
type=PATH msg=audit(1421820423.127:239): item=0 name="/dbfp/pgdatadata/postgresql.conf" inode=25
dev=08:21 mode=0100600 ouid=500 ogid=500 rdev=00:00 nametype=NORMAL
type=SYSCALL msg=audit(1421820423.128:240): arch=c000003e syscall=2 success=yes exit=3
a0=110dd20 a1=0 a2=0 a3=1 items=1 ppid=1179 pid=3251 auid=500 uid=500 gid=500 euid=500 suid=500
fsuid=500 egid=500 sgid=500 fsgid=500 tty=pts0 ses=1 comm="vim" exe="/usr/bin/vim" key=(null)
type=CWD msg=audit(1421820423.128:240):  cwd="/home/vagrant"
type=PATH msg=audit(1421820423.128:240): item=0 name="/dbfp/pgdatadata/postgresql.conf" inode=25
dev=08:21 mode=0100600 ouid=500 ogid=500 rdev=00:00 nametype=NORMAL
type=USER_CMD msg=audit(1421820425.220:241): user pid=3253 uid=500 auid=500 ses=1
msg='cwd="/home/vagrant" cmd=7461696C202F7661722F6C6F672F61756469742F61756469742E6C6F67
terminal=pts/0 res=success'
type=CRED_ACQ msg=audit(1421820425.221:242): user pid=3253 uid=0 auid=500 ses=1
msg='op=PAM:setcred acct="root" exe="/usr/bin/sudo" hostname=? addr=? terminal=/dev/pts/0
res=success'
type=USER_START msg=audit(  :243): user pid=3253 uid=0 auid=500 ses=1 msg='op=PAM:session_open
acct="root" exe="/usr/bin/sudo" hostname=? addr=? terminal=/dev/pts/0 res=success'
```

4.6. 出力したログの保全（改ざん防止）

対応する PCI DSS 要件	10.5.2、10.5.5

PostgreSQL には、リアルタイムにサーバログの改ざんを防止する仕組みはありません。サーバログの改ざんを防止するためには、別途 OS や別のツールを組み合わせて利用することが必要になります。

ここでは、サーバログの改ざん防止について、下記 3 つの観点で実施すべきことをまとめます。

1)改ざんされにくくする
2)改ざんを検出する
3)改ざんされていないことを保証する

4.6.1. 改ざんされにくくする

ログへのアクセスを制限することで改ざんされにくくすることができます。例えば、書き込み権限はオーナーのみとするなどです。

PostgreSQL では、log_file_mode でログファイルの権限を変更できます。
「log_file_mode = 0000」と設定することで、PostgreSQL の起動ユーザでも読み込み権限がない状態(書き込み権限のみある)にすることができます。

特に、log_destination、logging_collector、log_directory の設定で、データベースクラスタと異なる場所にログを出力している場合には、log_file_mode で適切な権限を与えてください。

4.6.2. 改ざんを検出する

仮に改ざんが発生したとしても、そのことを検出する仕組みを導入しておくことも重要になります。
これは、auditd など OS の機能で代替することができます。
詳細は、「4.5 DBMS 一般情報へのアクセス情報の取得」を参照してください。

4.6.3. 改ざんされていないことを保証する

改ざんの検出が困難な場合、そのログが改ざんされていないことを保証する仕組みを導入することで、ログの信ぴょう性を担保することができます。例えば、別のツールと組み合わせて保証するといったことです。
ここでは、OpenSSL の電子署名を用いた方法を例示します。下記の例で使用する鍵の信ぴょう性を担保するためには、電子証明書による管理がありますが、本書では割愛します。詳細は OpenSSL のドキュメント[20]等を参照してください。

(1)秘密鍵を作成する

```
$ openssl genrsa -out private.key
Generating RSA private key, 1024 bit long modulus
.....++++++
...++++++
e is 65537 (0x10001)
```

(2)公開鍵を作成する

```
$ openssl rsa -in private.key -pubout -out public.key
writing RSA key
```

20 OpenSSL Documents https://www.openssl.org/docs/

(3)電子署名を作成する

(1)で作成した秘密鍵をキーにして、電子署名(postgresql-2015-02-27_000000.sig)を作成します。

```
$ sha1sum postgresql-2015-02-27_000000.csv | awk '{print $1}' |
                    openssl rsautl -sign -inkey private.key > postgresql-2015-02-27_000000.sig
```

(4)確認する

ログファイルを参照する前に、作成した電子署名との突き合わせを行い、改ざんの有無を確認できます。
まずはじめに、電子署名から元のログファイルから算出したハッシュ値を取得します。

```
$ openssl rsautl -verify -in postgresql-2015-02-27_000000.sig -pubin -inkey public.key
687361549494304ccbfc71339ebc3b7da356191f
```

(4-1)改ざんされていない場合

対象のログファイルからハッシュ値を算出し、(4)で取得したハッシュ値と一致すれば改ざんされていないことが分かります。

```
$ sha1sum postgresql-2015-02-27_000000.csv
687361549494304ccbfc71339ebc3b7da356191f  postgresql-2015-02-27_000000.csv
```

(4-2)改ざんされている場合

対象のログファイルからハッシュ値を算出し、(4)で取得したハッシュ値と異なる場合は、改ざんされていることが分かります。

```
$ sha1sum postgresql-2015-02-27_000000.csv
474033de0887bcd31f9629e72d2f3c75e9bd27ea  postgresql-2015-02-27_000000.csv
```

4.7. (参考)今後、監査のユースケースに求められる PostgreSQL の強化機能について

ここでは、改めて監査に求められる機能について触れ、一般的な商用監査製品を参考にしながら、今後 PostgreSQL(執筆時にでは最新のメジャーバージョンは PostgreSQL9.5) や関連ツールに実装が期待される監査機能の内容について検討を行いました。特に、ユーザが個別の設計が必要な項目を洗い出すことによって、必要な要件に対対応する作業ボリュームをイメージすることを目的にしています。

近年では商用の監査製品が PostgreSQL に対応を開始しているケースもありますので、商用の監視ソフトウェアを利用することも選択肢の一つとなるでしょう。

なお、Database セキュリティコンソーシアムでは DB 内部不正防止に必要となるガイドラインを公開されておりますので一読されることをお薦めいたします。

Database セキュリティコンソーシアム DB 内部不正防止ガイドライン
http://www.db-security.org/report/ag_seika.html

4.7.1. 監査データ取得機能の強化について

監査データの収集では、悪意を持ったユーザがデータベースへの活動(設定の変更、データの取得など)をした場合に、それらのすべての活動を監査記録として確実に保管することが重要です。そのため、DB としての利用者情報や、サーバ設定などには、情報が変更された都度を記録・格納される枠組みを準備しておくことが必要になります。

上記観点を元に、DB サーバとしての監査情報として必要となる可能性がある項目とその対応状況を以下の表に整理しました。なお、これらの出力情報の項目については商用のデータベース監査ツールを参考に掲載を行っています。

なお、PostgreSQL サーバの機能により監査データの出力の仕組みが不足しているものについては、監査データの重要性を精査し、個別対応の検討を行うことが必要となります。

表 4.4: データベース監査に必要な監査データと PostgreSQL の対応状況

大項目	小項目	PostgreSQL サーバ側で情報出力が可能かどうか (PostgreSQL 9.5)	備考
クライアント情報	IP アドレス	○	PostgreSQL のサーバログへの出力(log_line_prefix または log_destination =csv_log)
	MAC アドレス	×	PostgreSQL の機能としては出力されないため、OS 側コマンドとの連携が必要
	クライアントホスト名	○	PostgreSQL のサーバログへの出力(log_line_prefix または log_destination =csv_log)
	クライアントの OS	×	PostgreSQL サーバからは出力されない
	アプリケーション側のユーザ特定	×	アプリケーション側のユーザを postgreSQL サーバ側から判断できる機構はない。特にコネクションプーリングを用いるシステムでは複数のアプリケーションユーザが1つの DB ユーザを共通で利用するケースがあり、DB ユーザだけでは判断ができない。 代替手段の一例としては、アプリケーション側で識別できる SQL 文を組み込むなどが考えられる。 なお、商用監査ツールでは、一部のビジネスアプリケーションのユーザを監査ログとして抽出する機能を持つものがある

	セッション時間	○	PostgreSQL のサーバログにセッション開始時刻とタイムスタンプが出力され、各ログ出力時のセッション時間の計算が可能（ただし、データとしての加工には工夫が必要） （log_line_prefix または log_destination =csv_log) 全体のセッション時間については log_connections および log_disconnections の設定で接続開始時間と終了時間が出力可能
	ログイン失敗	○	PostgreSQL のサーバログへの出力 (log_connections = on)
サーバ情報	サーバ IP	△	PostgreSQL のサーバログとしては出力されない。OS 側の情報を取得、変更履歴については都度情報を取得するなどの工夫が必要
	サーバポート	△	PostgreSQL のサーバログには出力されない システムカタログの pg_settings より取得が可能だが起動直後の定常的な取得の仕組みが必要。 なお、postgresql.conf では起動パラメータで変更が可能であるため不適
	サーバ名	△	OS 側の情報を取得、変更履歴については都度情報を取得するなどの工夫が必要
	サーバ OS	△	OS 側の情報を取得、変更履歴については都度情報を取得するなどの工夫が必要
	タイムスタンプ	○	PostgreSQL のサーバログへの出力される （log_line_prefix または log_destination =csv_log)
	セッション識別子 （プロセス ID）	○	PostgreSQL のサーバログへの出力される （log_line_prefix または log_destination =csv_log)
DB 情報	SQL 文	○	PostgreSQL のサーバログへの出力される (log_statement= all で設定) ただし、入力された SQL 文がそのまま単純に出力される。そのため、PREPARE や関数等で実行した内容はそのまま出力されてしまう。
	DB ユーザ名	○	PostgreSQL のサーバログへの出力される （log_line_prefix または log_destination =csv_log)
	DB バージョン	△	PostgreSQL のサーバログには出力されない SELECT version();等で取得する。変更履歴については都度情報を取得するなどの工夫が必要
	DB 側のエラー	○	PostgreSQL のサーバログへの出力される
	接続プロトコル	×	PostgreSQL のサーバログには出力されない
	SELECT 結果の件数確認	×	PostgreSQL のサーバログには出力されない
	トランザクション識別子	○	PostgreSQL のサーバログへの出力される （log_line_prefix または log_destination =csv_log)

（判定基準）
○：サーバログに随時出力され、その時点での内容が確認可能
△：PostgreSQL サーバログ以外の手段で取得が可能。該当時間の状態確認には工夫が必要
×：PostgreSQL の機能では出力できない

4.7.2. 監査出力データの範囲絞込み機能

監査記録は必要となるデータを全て確保することが重要ですが、元々利用する可能性がない不要な監査データまで取っておく必要はありません。必要以上に監査データを取得することは、その膨大なデータを格納するためのストレージの確保、保全のためのバックアップ運用の考慮、セキュリティインシデントに対する分析の準備など、データベースへの負荷だけでなく監査データに対する運用面の考慮についても負担が大きくなってしまいます。

そのため、事前に重要と思われる項目に監査データを絞り込んでおくことが効果的です。ここでは監査データを絞込む代表的な観点例を記載し、それらの対応状況を以下の表にまとめました。これらの観点は組み合わせて使うことで不適切と思われる操作を絞り込むことができ、出力された場合にアラートを即時発行するような運用も可能になります。また、[4.2.4 コミュニティでの pgaudit の開発動向]で紹介している通り、PostgreSQL に向けた監査用の拡張ツールがコミュニティによって開発されており、開発コミュニティ版 pgaudit の対応状況も併記しました。

表 4.5: データベース監査ログの取得絞込み観点例一覧

絞込み観点例	内容	PostgreSQL 標準機能	pgaudit[21] Ver 1.0.0
サーバログと監査データの分離	サーバログと監査データを別ファイルや別データとして出力し、監査を行いやすくする	×	○
データベースオブジェクト	個人情報やクレジットカード情報など重要なテーブルやカラムなどのオブジェクトに絞り込む。	×	○
データベースユーザ	管理者権限をはじめとする特別なユーザのデータベースに対する操作を記録しておく。	×	×
SQL 文	特定の命令のみ（SELECT, UPDATE, DELETE, TRUNCATE, ALTER USER…等）の監査データを取得する。 これはデータベースオブジェクトを指定すること併用して利用することでさらに有効になると考えられる。	×	△ READ,WRITE,FUNCTION,ROLE,DDL,MISC といったクラス（カテゴリ）単位で指定することは可能であるが文ごとの柔軟な指定はできない。
SELECT 件数	一定以上の大量なデータ取得や変更を検知し、データの菱栄や改ざんを検知する。	×	×
時間帯	特定の時間帯のみ監査ログを出力する。例えば通常のシステム稼働時間外のデータベースへの操作を記録しておくなどが考えられる。	×	×

4.7.3. 監査データの別サーバへの出力（改ざん対策など）

監査データは確実に正しく記録されることが重要な項目の一つですが、PostgreSQL では OS 上から見えるファイルシステムにログファイルとして監査データを出力することが一般的な方法の一つです。ファイルに出力した場合は、OS 上の管理者権限を持つユーザや PostgreSQL サーバの管理者権限を持つユーザには監査データの改ざんが基本的に可能となってしまいます。

具体的には[4.6 出力したログの保全（改ざん防止）]では、出力が完了したログファイルに対してアクセス権限をつける方法や監査証跡をつける方法を紹介していますが、PostgreSQL スーパユーザにはファイルのオーナー権限や書き込み権限は残っており、通常のセキュリティ設定では設定変更によるアクセスが可能である点について問題が残っています。また、監査証跡を取得する方法についても、データ変更がないことが確定したうえでないと証跡を確保できないため、監査証跡を残す前に更新されたものについては不正があったかどうかが判断できないなど課題は残っています。

21 PostgreSQL Audit Extension https://github.com/pgaudit/pgaudit

改ざんの1つの対策としてデータベースマシンの権限と監査データのアクセス権限を分離することが挙げられます。具体的な手法としては、syslog 系のツールを用いる方法があります。複数のデータベースシステムの監査データを1つの監査リポジトリに集約することで、セキュリティ管理者が一元管理し運用管理の面でも向上する効果もあります。

また、rsyslog 等を用いれば別の PostgreSQL サーバにログデータを格納することが可能になります。データベースに格納しておくことで、監査者がファイルベースではなくデータベースに対する検索機能（SQL 等）を利用することで、分析を行いやすくするというメリットもあるでしょう。

表 4.6: データベース監査データの集約によるメリット

検討項目	内容	ファイルへの出力	rsyslog でのリモート出力
改ざんへの対応	個人情報やクレジットカード情報など重要なテーブルやカラムなどのオブジェクトに絞り込む。	×	○
集約による運用改善	複数の PostgreSQL サーバの監査データを 1 か所に集約し、検索の仕組みやバックアップなど運用を行いやすくする	×	○
監査データの分析	監査データから全体の傾向や問題点の検出を行うための分析機能をもつ	×	△ （PostgreSQL 等にログデータを格納することが可能で SQL により検索可能）

4.7.4. 監査ポリシーの設定と分析・アラート出力の仕組み

不審なアクセスがあった場合、取得した監査データから管理者に不正を警告として通知する仕組みはセキュリティを確保するために重要な要件です。このために監査用の製品としては監査ポリシーを設定し、その監査ポリシーから外れたデータを監視し、メール等で通報を行う仕組みが必要です。

（例）監査ポリシーの例

- ログイン認証が 3 回連続エラーになった場合
- 管理者アカウントでのデータベースへのアクセスがあった場合
- サービス開始後に DDL が発行された場合
- PostgreSQL やサーバに対する設定変更が行われた場合
- 意図しない大量なデータアクセスがあった場合、
- サービス稼働時間以外へのデータの更新処理

現状、サーバログから上記を検出するためには、管理者がログの内容を確認して、状況を把握する必要があります。不正アクセスについては出力ログ 1 行で判断できるため、[4.24 不正アクセスのメール通知]や[4.25 不正アクセスの SNMP 通知]で通知手法が記載されている通り、ログインの失敗を検出/通知が可能です。しかし、複数のレコードにまたがる分析が必要な場合は、PostgreSQL の技術者による判断が必要であり、リアルタイムでの通知については難しい面があります。

例えば、ログイン認証でも毎回不正の情報を通知するのではなく、短期間に複数回ログイン認証がエラーになった場合を検知するなど高度な判定が求められます。

この様に実際に収集したデータを分析し、その監査データから何が問題であるか、そしてどのように検知をするかというナレッジと方法論がまだ確立できていません。分析・通知のためのツールやナレッジ等が今後整備されていくことが期待されます。

4.8. CSV サーバログのテーブルへのロード方法

対応する PCI DSS 要件	10.2, 10.3, 10.6

4.8.1. サーバログ検索の問題点

サーバログを検索するには、通常は grep などの OS コマンドを使用します。
シンプルな検索であれば問題ありませんが、以下の様な複雑な条件になるとプログラミングが必要となります。

- 1 レコードが複数行から構成される場合
  アプリケーションから実行される SQL は複数行で記述されることが一般的です。
  grep コマンドではキーワードが存在する行しか認識できません。
- レコード間の関連を確認する場合
  接続失敗がある一定期間に一定回数以上発生したかどうかを確認する場合など、レコード間の関連を条件とする必要があります。

サーバログを SELECT 文で検索できると便利です。
以下の方法で実現が可能です。ただし何れの方法もサーバログを CSV 形式で出力することが必要です。
方法 1: サーバログを定期的にテーブルにロード
方法 2: サーバログを外部表として定義

CSV 形式は log_line_prefix によるカスタマイズはできませんが、以下のメリットがあります。
・ 出力形式が固定であるため、テーブルへのロードや awk コマンドでの検索に便利
・ 全ての項目が出力されるため、取得漏れの懸念が無い

4.8.2. 設定例

ここでは方法 1 で説明します。
PostgreSQL のマニュアル（18.8. エラー報告とログ取得）にも説明がありますのでご参照ください。

1. postgresql.conf の設定

表 4.7: 設定が必要なパラメータ

項目	概要
log_destination	'csvlog' または 'stderr,csvlog'

2. reload による即時反映
   上記パラメータはリロードにより即時に反映させることができます。
   次に発行される SQL から CSV ログに出力されます。（既存セッションに対しても有効です）。
   (実行例)

```
$ pg_ctl reload
server signaled
```

   パラメータに変更があった場合はサーバログに変更の旨が出力されます。
   (出力例)

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "log_statement" changed to "all"
```

3. サーバログをロードするテーブルの作成
   以下のテーブルを作成します。

表 4.8:　CSV サーバログの出力項目

項番	項目名	内容
1	log_time	接続時刻
2	user_name	接続した DB ユーザ名
3	database_name	接続先のデータベース名
4	process_id	OS のプロセス ID
5	connection_from	クライアントホスト名または IP アドレス
6	session_id	セッション ID
7	session_line_num	各セッションまたは各プロセスのログ行の番号
8	command_tag	コマンド種別　例）認証エラーは、"authentication"
9	session_start_time	セッション開始時間
10	virtual_transaction_id	仮想トランザクション ID
11	transaction_id	トランザクション ID
12	error_severity	重要度　例）INFO、NOTICE、WARNING、ERROR、LOG、FATAL、PANIC
13	sql_state_code	SQLSTATE（SQL の戻り値）
14	message	メッセージ
15	detail	message を補足する詳細なメッセージ
16	hint	ヒント　(エラーの場合に、問題箇所を示唆)
17	internal_query	内部 SQL
18	internal_query_pos	内部 SQL 位置
19	context	コンテキスト
20	query	アプリケーションなどによって明示的に発行された SQL 文
21	location	ロケーション
22	application_name	アプリケーション名（明示的に設定する事で判別が容易に）

テーブル作成文

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
```

```
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

4. CSV サーバログをテーブルにロード

   COPY FROM コマンドで CSV ファイルを postgres_log テーブルにロードします。

   (実行例)　$PGDATA='/postgres/data' とする

```
=# COPY postgres_log FROM '/postgres/data/pg_log/postgresql-2015-01-14_000000.csv' WITH
csv;
COPY 999
```

4.8.3. サーバログ情報の検出

例として、認証エラーが発生した日時とユーザ名を取得します。

条件: コマンド種別が"authentication"　かつ SQLSTATE が'00000'以外

```
=# SELECT log_time,
-#        user_name,
-#        sql_state_code
-# FROM   postgres_log
-# WHERE  command_tag = 'authentication'
-# AND    sql_state_code != '00000';
          log_time            | user_name |sql_state_code
------------------------------+-----------+--------------
 2015-01-14 00:18:12.635+09   | user1     |28000
 2015-01-14 00:18:12.635+09   | user2     |28000
 .. 以下略
```

4.9. 特定のクライアントからのアクセスを拒否する

対応する PCI DSS 要件	10.2, 10.3, 10.6

pg_hba.conf によるクライアント認証にて、特定のクライアントからのアクセスを拒否することができます。クライアントには、IP アドレス、ホスト名、CIDR 指定によるセグメント単位の指定が可能です。

4.9.1. 設定

先に記載した設定が優先されるため、例外的な設定を先に記載することで、ブラックリスト方式やホワイトリスト方式による柔軟な設定が可能です。

1. pg_hba.conf の設定

設定例 1)　ブラックリスト方式
IP アドレスが 192.168.150.* のクライアントは許可します（基本）。
ただし、192.168.150.1 は拒否します（例外）。

```
host    all    all    192.168.150.1/32    reject
host    all    all    192.168.150.0/24    md5
```

設定例 2)　ホワイトリスト方式
IP アドレスが 192.168.150.* のクライアントは拒否します（基本）。
ただし、192.168.150.1 は許可します（例外）。

```
host    all    all    192.168.150.1/32    md5
host    all    all    192.168.150.0/24    reject
```

2. reload による即時反映

リロードにより pg_hba.conf の設定を即時に反映させることができます。
以降の接続要求からクライアント認証の対象となります。

(実行例)

```
$ pg_ctl reload
server signaled
```

サーバログにリロードされた旨が出力されます。

(出力例)

```
LOG:  received SIGHUP, reloading configuration files
```

4.9.2. 不正アクセスの試行

許可されていないクライアントからデータベースへの接続を試行するとエラーとなり、サーバログに以下のような認証エラーのメッセージが出力されます。エラーの出力は log_min_messages の設定であり、log_connections および log_disconnections が有効でない場合でも出力されます。

(実行例)　クライアント認証で拒否に設定したクライアントから 接続要求がきた場合

```
$ psql -h 192.168.150.140  -U user1
psql: FATAL:  pg_hba.conf rejects connection for host "192.168.150.1", user "user1",
database "database1", SSL off
```

(サーバログ出力例)

```
FATAL:  pg_hba.conf rejects connection for host "192.168.150.1", user "user1", database
"database1", SSL off
```

表 4.9: サーバログ出力内容詳細

レベル	出力内容
FATAL	FATAL: pg_hba.conf rejects connection for host "192.168.150.1", user "user1", database "database1", SSL off クライアント認証での拒否設定による出力です。

(サーバログ出力例)　csvlog 形式の場合 (便宜上、空白行を挿入)

```
2015-02-05 19:34:26.507 JST,"user1","database1",12177,"192.168.150.1:57565",54f83132.2f91,
2,"authentication",2015-02-05 19:34:26 JST,1/17,0,FATAL,28000,"pg_hba.conf rejects
connection for host ""192.168.150.1"", user ""user1"", database ""database1"", SSL off"
,,,,,,,,,""
```

4.9.3. 不正アクセスの検出

grep コマンドで上記のキーワードを指定してサーバログを検索します。

(検出例)

```
$ grep "rejects connection" $PGDATA/pg_log/postgresql-*.log
```

4.10. OS ユーザと DB ユーザのマッピング（シンプルなマッピング）

対応する PCI DSS 要件	10.2, 10.3, 10.6

本節および次節の peer 認証の説明では、OS ユーザと DB ユーザを区別するため、ユーザ名を””で囲み、OS ユーザまたは DB ユーザを記載する表記とします。
例）”user1”OS ユーザ　/ “user1”DB ユーザ

peer 認証または ident 認証にて、OS ユーザ（アカウント）と DB ユーザを関連付けることができます。この機能により、管理者アカウントと一般アカウントの差別化を図ることができます。
例えば PostgreSQL の任意のデータベースに対して”user1”DB ユーザとして接続するには、まず”user1”OS ユーザにログインします。これによりスーパユーザによる接続は特定の OS ユーザに限定する事ができます。

表 4.10: 主な認証方式

認証方式	説明
trust	接続を無条件で許可（パスワード不要）
reject	接続を無条件で拒否
md5	クライアントに対して認証時に md5 暗号化パスワードを要求
password	クライアントに対して認証時に平文のパスワードを要求
peer	クライアントの OS ユーザと DB ユーザのマッピングを行う ローカル接続時のみ使用可能
ident	クライアント上の ident サーバに問合せ、OS ユーザ名が要求された DB ユーザ名と一致するか確認 TCP/IP 接続でのみ使用可能であり、ローカル接続では peer 認証が使用される
ldap	LDAP サーバを使用して認証

接続可能な DB ユーザを、OS ユーザ名と同一名の DB ユーザに限定します。

4.10.1. 設定

1. pg_hba.conf の設定

表 4.11: シンプルな peer 認証におけるクライアント認証設定例）

項目	概要
TYPE	local
DATABASE	データベースへの接続を制限する場合に設定。設定例では”all”。
USER	接続を制限する OS ユーザを指定。設定例では”user1”。
ADDRESS	ローカルであるため、未指定（空白）。
METHOD	peer　（オプションなし）

設定例）　設定① peer 認証により、”user01”DB ユーザへは”user1”OS ユーザのみ接続を許可

```
# TYPE    DATABASE      USER       ADDRESS          METHOD
local     all           user1                       peer        ①
```

2. reload による即時反映
   リロードにより pg_hba.conf の設定を即時に反映させることができます。
   実行例）

```
$ pg_ctl reload
server signaled
```

サーバログにリロードされた旨が出力されます。

出力例）

```
LOG:  received SIGHUP, reloading configuration files
```

4.10.2. 接続の試行とサーバログの確認

以降の接続から pg_hba.conf の制限が有効になります。上記設定における成功例と失敗例を記載します。

(実行例 1：成功例） ”user1”OS ユーザが”user1”DB ユーザにて接続を試行

```
$ id
uid=1005(user1) gid=1002(group2) 所属グループ=1002(postgres)

$ psql -U user1 postgres
=#                                                              -- 認証成功
```

(実行例 2：失敗例） ”user2”OS ユーザが “user1”DB ユーザにて接続を試行

```
$ id
uid=1005(user2) gid=1002(group1) 所属グループ=1002(group1)

$ psql -U user1 postgres
psql: FATAL:  Peer authentication failed for user ”user1”          --認証失敗
```

(サーバログ出力例） stderr 形式の場合、以下の 4 行が出力される可能性があります。
3 行目と 4 行目はセットです。DETAIL(詳細メッセージ）がある場合は 2 行の出力となります。

```
LOG:  connection received: host=[local]
LOG:  provided user name (user1) and authenticated user name (user2) do not match
FATAL:  Peer authentication failed for user ”user1”
DETAIL:  Connection matched pg_hba.conf line 88: ”local   all   user1   peer”
```

表 4.12: 上記におけるサーバログ出力の説明

レベル	出力内容
LOG	connection received: host=[local] 認証要求を受けた事を示しています（接続は完了していません）。 log_connections パラメータが on の場合に出力されます。
LOG	provided user name (user1) and authenticated user name (user2) do not match 接続先 DB ユーザ(user1)と接続元 OS ユーザ(user2)が一致していない事を示しています。 log_min_messages パラメータのレベルが log を含む場合に出力されます。
FATAL	Peer authentication failed for user "user1" peer 認証によるエラーである旨が出力されます。SQLSTATE は 28000 です。 log_min_messages パラメータのレベルが fatal を含む場合に出力されます。
-	Connection matched pg_hba.conf line 85: "local all user1 peer" pg_hba.conf の行数と設定内容が出力されます。直前の FATAL メッセージの補足出力です。

pg_hba.conf の設定行数が出力されるため、peer 認証設定が複数ある場合でも切り分けが容易です。

4.10.3. 認証エラーの検出

grep コマンドで上記のキーワードを指定します。

検出例）

```
$ grep ” Peer authentication failed ” $PGDATA/pg_log/postgresql-*.log
```

4.11. OS ユーザと DB ユーザのマッピング（柔軟なマッピング）

対応する PCI DSS 要件	10.2, 10.3

pg_ident.conf を使用する事で、OS ユーザ名と DB ユーザの柔軟なマッピングを可能にします。

4.11.1. 設定

•pg_ident.conf の設定

表 4.13: pg_ident.conf の設定項目

項目	概要
MAPNAME	任意のマップ名を指定。
SYSTEM-USERNAME	OS ユーザ名を指定。正規表現で指定可能。
PG-USERNAME	DB ユーザ名を指定。

(設定例) 同一マップによる複数行の設定

```
# MAPNAME      SYSTEM-USERNAME      PG-USERNAME
map1           /user[1-2]           user1          #設定①
map1           user3                user2          #設定②
```

設定① ”user1”/”user2”OS ユーザ から”user1”DB ユーザにてデータベースに接続可能
SYSTEM-USERNAME で正規表現を使用する場合は、先頭に”/”を付与
設定② “user3”OS ユーザは、”user2”DB ユーザにてデータベースに接続可能

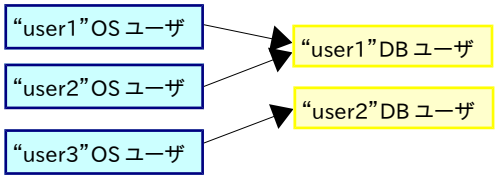


図 4.2: OS ユーザと DB ユーザの関係

•pg_hba.conf の設定

表 4.14: 柔軟な peer 認証のクライアント認証設定例）

項目	概要
TYPE	local を設定。
DATABASE	データベースへの接続を制限する場合に設定。 以下の例では、all を設定。
USER	接続を制限する OS ユーザを設定。 以下の例では、user1,user2 を設定。
ADDRESS	ローカルであるため、未指定（空白）。
METHOD	peer および map オプションとしてマップ名を設定。

(設定例 1) USER 項目には、map1 に該当する”user1”,”user2”DB ユーザをカンマ区切りで指定

```
# TYPE    DATABASE      USER              ADDRESS       METHOD
local       all         user1,user2                     peer map=map1        ①
```

設定① ”user1”,”user2”DBユーザへ接続する場合は、マッピング map1 のルールで認証

•reload による即時反映

リロードにより pg_hba.conf の設定を即時に反映させることができます。
以降の接続から pg_hba.conf の制限を受けます。

(実行例)

```
$ pg_ctl reload
server signaled
```

サーバログにリロードされた旨が出力されます。
(出力例)

```
LOG:  received SIGHUP, reloading configuration files
```

4.11.2. 接続の試行及びサーバログの出力確認

上記設定における成功例と失敗例を記載します。

実行例 1: 成功例) ”user2”OS ユーザが “user1”DB ユーザ にて接続を試行

```
$ id
uid=1007(user2) gid=1002(postgres) 所属グループ=1002(postgres)

$ psql -U user1 database1
=#                                                              -- 認証成功
```

(実行例 2: 失敗例) ”user2”OS ユーザが “user2”DB ユーザ にて接続を試行

```
$ id
uid=1007(user2) gid=1002(group1) 所属グループ=1002(group1)

$ psql -U user2 database1
psql: FATAL:  Peer authentication failed for user ”user2”          --認証失敗
```

(サーバログ出力例) stderr 形式の場合、以下の 4 行が出力される可能性があります。
3 行目と 4 行目はセットです。DETAIL(詳細メッセージ)がある場合は 2 行の出力となります。

```
LOG:  connection received: host=[local]
LOG:  no match in usermap ”map1” for user ”user2” authenticated as ”user2”
FATAL:  Peer authentication failed for user ”user2”
DETAIL:  Connection matched pg_hba.conf line 88: ”local   all    user1,user2,user3
peer map=map1 ”
```

表 4.15: サーバログ出力内容詳細

レベル	出力内容
LOG	connection received: host=[local] 認証要求を受けた事を示しています(接続は完了していません)。 log_connections パラメータが on の場合に出力されます。
LOG	no match in usermap "map1" for user "user2" authenticated as "user2" 接続先 DB ユーザ user2 と接続元 OS ユーザ user2 が map1 のマッピング設定に一致していない事を示しています。log_min_messages パラメータが log 以上の場合に出力されます。
FATAL	Peer authentication failed for user "user2" peer 認証によるエラーである旨が出力されます。SQLSTATE は 28000 です。 log_min_messages パラメータが fatal 以上の場合に出力されます。
-	Connection matched pg_hba.conf line 88: "local all user1,user2,user3 peer map=map1 " pg_hba.conf の行数と設定内容が出力されます。直前の FATAL メッセージの補足出力です。

4.11.3. 検出

(検出例)　grep コマンドで上記のキーワードを指定します。

```
$ grep ” Peer authentication failed ” $PGDATA/pg_log/postgresql-*.log
```

pg_hba.conf の設定行数が出力されるため、peer 認証設定が複数ある場合でも切り分けが容易です。

4.12. パスワードポリシーの設定

対応する PCI DSS 要件	8.5.10, 8.5.11

PostgreSQL 本体には、パスワードポリシーを設定する機能はありませんが、contrib として標準で付属している passwordcheck を使用することで、以下のパスワードポリシーを設定することができます。このポリシーに違反するパスワードの設定を CREATE ROLE 文や ALTER ROLE 文にて試みると、エラーが発生します。

1. 8 文字以上である事
2. アルファベット(A～Z, a～z) およびアルファベット以外の文字をそれぞれ一文字以上使用する事
3. ユーザ名を含まない事

上記ポリシーは passwordcheck.c またはヘッダファイルを変更する事によりカスタマイズも可能です。
本機能はパスワードチェックフックを利用して実装されます。

またオプションとして、CrackLib を使用する事もできます。CrackLib の辞書に登録された文字列を含んだパスワードの設定を試みるとエラーとなります。

なお、より多機能なパスワードポリシーを求めるのであれば、LDAP と連携する方法（4.1 アカウントポリシー機能の実現）があります。パスワードポリシー以外にも、一定回数連続でパスワードを誤ると一定期間ロックされる機能などがあります。

4.12.1. 設定

設定方法を記載します。PostgreSQL 9.5 を使用しました。

1. passwordcheck のコンパイル
   PostgreSQL のソースツリーからコンパイルします。既にコンパイル済みであればこの手順は不要です。
   ここでは以下の方針です。
   - contrib 全体をコンパイルする方法もありますが、passwordcheck に限定します。
   - CrackLib は使用しません。

   （実行例）
   ```
   $ cd <ソースツリー>/contrib/passwordcheck
   $ make
   $ make install
   ```

2. shared_preload_libraries パラメータ の設定
   $PGDATA/postgresql.conf に以下を設定して PostgreSQL を再起動します。
   起動時にライブラリ passwordcheck.so が読み込まれます。

   （postgresql.conf の設定例）
   ```
   shared_preload_libraries = '$libdir/passwordcheck'
   ```

   （再起動の実行例）
   ```
   $ pg_ctl restart -m fast -w
   ```

これで passwordcheck の設定は終了です。

4.12.2. 基本動作確認

ユーザのパスワード変更時に、ポリシーに違反するパスワードの場合はエラーが発生することを確認します。

■パスワード長
パスワード長さが 8 文字未満の場合は本エラーが発生します。最初にチェックされる条件であるため、他のポリシー違反に抵触する場合でも本エラーが発生します。

（失敗例）ユーザ名を含むポリシー違反も含んでいるがパスワード長が短すぎるエラーが発生する。

```
=# ALTER ROLE user1 PASSWORD 'user1';
ERROR:  password is too short
```

（失敗例）長さ条件は満たしているが、アルファベットのみ

```
=# ALTER ROLE user1 PASSWORD 'abcdefgh';
ERROR:  password must contain both letters and nonletters
```

（失敗例） 長さ条件は満たしているが、数字のみ（アルファベットのみと同じメッセージ）

```
=# ALTER ROLE user1 PASSWORD '12345678';
ERROR:  password must contain both letters and nonletters
```

（失敗例）長さ条件および文字種条件を満たすが、ユーザ名を含む

```
=# ALTER ROLE user1 PASSWORD 'user1234';
ERROR:  password must not contain user name
```

（成功例）長さ条件および文字種条件を満たし、ユーザ名を含まないのであれば、成功

```
=# ALTER ROLE user1 PASSWORD 'abcd1234';
ALTER ROLE
```

4.12.3. CrackLib 動作確認

CrackLib を使用する方法を紹介します。ここでは CrackLib 2.9.0 を使用します。

1. CrackLib の導入
   CrackLib の Tar ボールをサーバにアップしてインストールします。
   configre にてオプションで-with-default-dict を指定します。
   後工程で辞書ファイルを作成するディレクトリのデフォルトとなります。辞書ファイル作成時にディレクトリを指定する事もできるため、必須ではありません。

   （実行例）

```
# tar xvzf cracklib-2.9.0.tar.gz
# cd cracklib-2.9.0
# ./configure -with-default-dict=/usr/lib64/cracklib_dict
# make
# make install
```

   '/usr/lib64/cracklib_dict'と指定すると、辞書作成時に '/usr/lib64'ディレクトリに 'cracklib_dict.hwm','cracklib_dict.pwd','cracklib_dict.pwi'の 3 ファイルが作成されます。

2. 辞書単語帳のカスタマイズ
   辞書ファイル(/usr/local/share/cracklib/cracklib-small)を必要に応じてカスタマイズします。
   テキストファイルであるためエディタにて修正が可能です。

```
# vi /usr/local/share/cracklib/cracklib-small
```

任意の場所に追記します。ここでは末尾に”PGECONS2015”を追記します。

```
～　略　～
zurich
zxcvbn
zxcvbnm
zygote
PGECONS2015  ---末尾に追記
```

3. 辞書ファイルの作成

   create-cracklib-dict コマンドで辞書ファイルを作成します。

   configure で指定したディレクトリに作成されていることを確認します。

（実行例）

```
■作成
# /usr/local/sbin/create-cracklib-dict  /usr/local/share/cracklib/cracklib-small
49641 49641

■確認
# ls -l /usr/lib64/cracklib_dict.*
-rw-r--r--. 1 root root   1024  1月  6 23:24 2016 /usr/lib64/cracklib_dict.hwm
-rw-r--r--. 1 root root 235429  1月  6 23:24 2016 /usr/lib64/cracklib_dict.pwd
-rw-r--r--. 1 root root  12424  1月  6 23:24 2016 /usr/lib64/cracklib_dict.pwi
```

4. passwordcheck の再コンパイル

<ソースツリー>/contrib/passwordcheck/Makefile の修正

（修正前）CrackLib は使用されないようにコメントアウトされている

```
# uncomment the following two lines to enable cracklib support
# PG_CPPFLAGS = -DUSE_CRACKLIB '-DCRACKLIB_DICTPATH="/usr/lib/cracklib_dict"'
# SHLIB_LINK = -lcrack
```

（修正後）コメントアウトを外し、辞書ファイルのパスを修正

CRACKLIB_DICTPATH には configure で指定したパスを指定します。

```
# uncomment the following two lines to enable cracklib support
PG_CPPFLAGS = -DUSE_CRACKLIB '-DCRACKLIB_DICTPATH="/usr/lib64/cracklib_dict"'
SHLIB_LINK = -lcrack
```

（再コンパイル実行例）PostgreSQL を停止後、再コンパイルを行います。

```
$ pg_ctl stop -m fast -w

$ cd <ソースツリー>/contrib/passwordcheck
$ make uninstall
$ make clean
$ make
$ make install

$ pg_ctl start
```

5. 動作確認

   辞書単語帳に登録したパスワードにてエラーが発生することを確認します。

（失敗例）

```
=# ALTER ROLE user1 PASSWORD 'PGECONS2015';
ERROR:  password is easily cracked
```

4.12.4. エラーメッセージ一覧

passwordcheck では以下のエラーが発生する可能性があります。

表 4.16: passwordcheck のエラーメッセージ一覧

エラーメッセージ	内容
password encryption failed	パスワードが md5 暗号化されている場合
password is too short	パスワードが平文でパスワード長が 8 文字未満の場合
password must not contain user name	パスワードが平文でパスワードがユーザ名を含んでいる場合
password must contain both letters and nonletters	パスワードが平文で、パスワードがアルファベットのみ、またはアルファベット以外のみの場合
password is easily cracked	パスワードが平文で パスワードが CrackLib 辞書に登録がある場合

4.12.5. 注意点

1. ネットワーク経由で暗号化されたパスワードが送信された場合は passwordcheck ではエラーが発生します。パスワードの設定はローカルで設定します。

2. CREATE ROLE 文でパスワードの設定、または ALTER ROLE でパスワードの変更時にチェックされます。passwordcheck が有効化される前に設定されたパスワードはチェックされません。

4.13. パスワードの総当たり攻撃対策

対応する PCI DSS 要件	8.5.13

auth_delay はパスワードの総当たり攻撃対策として実装された標準の contrib です。
機能はシンプルであり、認証エラーの報告を行う前に指定した時間だけ停止(sleep)させるものです。
認証エラーの度に指定時間待機する事で、大量の接続試行に対して時間を稼ぐことが出来ます。
本機能はクライアント認証フックにて実装されています。

4.13.1. 設定

以下に設定方法を記載します。PostgreSQL 9.5 を使用しました。

1. auth_delay のコンパイル
   PostgreSQL のソースツリーからコンパイルします。既にコンパイル済みであればこの手順は不要です。
   ここでは以下の方針です。
   - contrib 全体をコンパイルする方法もありますが、auth_delay に限定します。

   （実行例）

```
$ cd <ソースツリー>/contrib/auth_delay
$ make
$ make install
```

2. パラメータの設定
   postgresql.conf に以下のパラメータを設定します。
   auth_delay.milliseconds は auth_delay 専用のパラメータです。

表 4.17: 設定が必要なパラメータ

パラメータ	設定値
shared_preload_libraries	auth_delay のライブラリを指定。 設定例）'auth_delay' 変更の反映には PostgreSQL の再起動が必要。
auth_delay.milliseconds	指定されたミリ秒数認証エラーを返す前に待機（カスタムパラメータ） 設定例）500　（500 ミリ秒） 変更の反映はリロードで可能

auth_delay の有効化/無効化は shared_preload_libraries の反映であるため再起動が必要ですが、遅延時間の変更は auth_delay.milliseconds の反映であるため、リロードで可能です。
ただし auth_delay.milliseconds を 0 に設定することで、リロードによる実質的な無効化は可能です。

(postgresql.conf の設定例)

```
shared_preload_libraries = 'auth_delay'          # (change requires restart)
auth_delay.milliseconds = 500
```

```
$ pg_ctl restart -m fasgt -w
```

これで auth_delay の設定は終了です。

4.13.2. 動作確認

パスワード認証のパスワード誤りによるエラーを発生させ、設定した遅延が発生することを確認する。

表 4.18: 動作確認用の設定

パラメータまたは環境変数	設定値
PGPASSWORD	設定値）bad_password パスワード認証に対してパスワード入力を不要にするために設定（時間計測のため） 誤ったパスワード
auth_delay.milliseconds	設定値）10000（10 秒） 遅延を分かりやすくするため、大きい値を設定

（実行例） psql の前後で10秒の差があるのが分かる。エラーが出力されるまでに10秒を要している。

```
$ pg_ctl reload
$ export PGPASSWORD=bad_password

$ date; psql -U user_a -h host1; date
2016 年  1 月  7 日 月曜日 10:00:00 JST
psql: FATAL:  password authentication failed for user "user_a"
2016 年  1 月  7 日 月曜日 10:00:10 JST
```

4.13.3. 注意点

1. DOS 攻撃を防ぐためのものでは無いことに注意してください。認証エラーを待たせ、コネクションスロットを消費させるため、むしろ DOS 攻撃を増長させるかもしれません。

4.14. 不正アクセスのチェック(パスワード攻撃の検出)

対応する PCI DSS 要件	10.2, 10.3

接続失敗回数が一定期間に想定以上ある場合、パスワード辞書攻撃の可能性を疑い、認証エラーを確認します。

4.14.1. 認証エラー一覧

以下に主な認証エラーを記載します。METHOD 列 はクライアント認証 pg_hba.conf の認証方式の指定です。

表 4.19: 主な認証エラー一覧

項番	原因	METHOD	サーバログに出力されたメッセージ	SQLSTATE
1	pg_hba.conf に該当する設定が無い	--	FATAL: no pg_hba.conf entry for host "[local]", user "xxxxx", database "xxxxx", SSL off	28000
2	データベースが存在しない	任意	FATAL: database "xxxxx" does not exist	3D000
3	ロールが存在しない	任意	FATAL: role "xxxxx" does not exist	28000
4	拒否設定	reject	FATAL: pg_hba.conf rejects connection for host "[local]", user "xxxxx", database "xxxxx", SSL off	28000
5	peer 認証エラー	peer	FATAL: Peer authentication failed for user "xxxxx"	28000
6	パスワードミス　または パスワードの有効期限切れ	password md5	FATAL: password authentication failed for user "xxxxx"	28P01
7	接続制限数オーバー	任意	FATAL: sorry, too many clients already	53300

パスワード辞書攻撃の可能性があるのは、項番 6 のパスワード不正です。
なおパスワード不正とパスワードの有効期限切れは区別ができないのでご注意下さい。

4.14.2. 認証エラーとログ出力例

以下に主な認証エラーを記載します。

(接続失敗例) user1 ユーザ(md5 認証) で接続を試みた際にパスワード入力をミス

```
$ psql -U user1 database1
Password for user user1:
psql: FATAL:  password authentication failed for user ”user1”
```

(stderr サーバログ出力例) 以下の 3 行が出力されます。

```
LOG:  connection received: host=[local]
FATAL:  password authentication failed for user ”user1”
DETAIL:  Connection matched pg_hba.conf line 89: ”local   all   user1    md5”
```

表 4.20: サーバログ出力内容詳細

レベル	出力内容
LOG	connection received: host=[local] 認証要求を受けた事を示しています(接続は完了していません)。 log_connections パラメータが on の場合に出力されます。
FATAL	password authentication failed for user "user1" パスワード認証エラーである旨が出力されています。SQLSTATE は “28P01” log_min_messages パラメータが fatal 以上の場合に出力されます。
-	Connection matched pg_hba.conf line 89: "local all user1 md5" pg_hba.conf の行数と設定内容が出力されます。直前の FATAL メッセージの補足出力です。

(csvlog サーバログ出力例) 以下の２行が出力されます。 便宜上、空白行を挿入。

```
2015-02-04 17:05:54.204 JST,,,20157,"",54d1d2e2.4ebd,1,"",2015-02-04 17:05:54 JST,,0,LOG,
00000,"connection received: host=[local]",,,,,,,,,""

2015-02-04 17:05:54.205 JST,"user1","database1",20157,"[local]",54d1d2e2.4ebd,2,
"authentication",2015-02-04 17:05:54 JST,1/3,0,FATAL,28P01,"password authentication failed
for user ""user1""","Connection matched pg_hba.conf line 89: ""local   all
user1       md5""",,,,,,,,,""
```

4.14.3. 認証エラーの検出

postgres_log テーブルを参照します。postgres_log テーブルの内容については 4.6.2 項をご参照ください。

検出例 1: 集計情報) 1 分間で 100 回以上のパスワード不正の発生を検出します。

条件: message 列にパスワード不正のキーワードを含む

集計: 毎分

グループ条件: 件数が 100 以上

```
=# SELECT to_char(log_time,'YYYY-MM-DD HH24:MI') AS log_time,
-#        count(*) AS cnt
-# FROM   postgres_log
-# WHERE  message LIKE '%password authentication failed%'
-# GROUP  BY to_char(log_time,'YYYY-MM-DD HH24:MI')
-# HAVING count(*) >= 100;

 log_time        |   cnt
-----------------+------
2015-01-14 18:48 |   931
```

2015-01-14 18:48 の 1 分間で、931 回のパスワード不正が発生しています。

サーバログから上記エラーの個別情報を確認するなどの対処を行います。

検出例 2: 個別情報)

該当日時の個別情報を取得します。

条件:ログイン時間に対して分精度で該当日時を指定

```
=# SELECT log_time,
-#        user_name,
-#        database_name,
-#        connection_from,
-#        message
-# FROM   postgres_log
-# WHERE  to_char(log_time,'YYYY-MM-DD HH24:MI') = '2015-01-14 18:48';

-[ RECORD 1 ]---+-----------------------------------
log_time        | 2015-01-14 18:48:03.051+09
user_name       |
database_name   |
connection_from |
message         | connection received: host=[local]
-[ RECORD 2 ]---+-----------------------------------
log_time        | 2015-01-14 18:48:03.051+09
user_name       | user1
database_name   | database1
connection_from | [local]
message         | password authentication failed for user "user1"
         以下略
```

上記の様に、「認証要求」と「パスワード不正」がセットで多数出力されます。

4.15. 不正アクセスのチェック（SQL 文の発行を検知(DDL 含む)）

対応する PCI DSS 要件	10.2, 10.3

発行された SQL 文のを行います。
SQL にはそれぞれリスクがあります。

- SELECT / COPY TO : 情報漏洩
- INSERT / UPDATE / DELETE : 情報改ざん
- TRUNCATE / COPY FROM : 情報改さん

4.15.1. 設定例

サーバログに必要な情報を出力するために以下の設定を行います。

1. postgresql.conf の設定

表 4.21: SQL をサーバログに出力するための監査設定例）

項目	概要
log_statement	全ての SQL を取得するため、'all' を設定 node : 取得しない ddl : DDL 文(CREATE / ALTER /DROP など)を取得 mod : 全ての DDL 文および更新 (UPDATE/DELETE/INSERT/TRUNCATE/COPY FROM)を取得 all : 全ての SQL 文を取得

2. リロードによる即時反映

上記パラメータはリロードにより即時に反映させることができます。
既存セッションに対しても有効であり、次に発行される SQL からサーバログに出力します。
(実行例)

```
$ pg_ctl reload
server signaled
```

パラメータに変更があった場合はサーバログに変更の旨が出力されます。
(出力例)

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "log_statement" changed to "all"
```

4.15.2. SQL 実行及びサーバログの確認

以下の SQL を実行します。
SELECT / DELETE / ALTER TABLE (DDL)

```
=# SELECT count(*) FROM pgbench_accounts ;                                    -- SELECT 文
 count
--------
 100000
(1 行)

=# DELETE FROM pgbench_accounts ;                                            -- DML 文
DELETE 100000

=# ALTER TABLE pgbench_accounts DROP CONSTRAINT pgbench_accounts_pkey ;       -- DDL 文
ALTER TABLE
```

(サーバログ出力例)

上記 SQL が出力されています。出力タイミングは SQL が発行された時点です(SQL 完了を待ちません)。

```
LOG:  statement: SELECT count(*) FROM pgbench_accounts ;
LOG:  statement: DELETE FROM pgbench_accounts ;
LOG:  statement: ALTER TABLE pgbench_accounts DROP CONSTRAINT pgbench_accounts_pkey ;
```

4.15.3. 実行済み SQL の検出

検出例: ALTER TABLE 文の発行頻度を調査
条件 : message 列に文字列"ALTER TABLE"を含む
集計 : 日別、データベース別

```
=# SELECT database_name,
-#        log_time::date AS log_date ,
-#        count(*) AS cnt
-# FROM   postgres_log
-# WHERE  upper(message) LIKE '%ALTER TABLE%'
-# GROUP  BY database_name,log_date;

 database_name |  log_date  | cnt
---------------+------------+-----
 database1     | 2015-01-26 |   3
 database2     | 2015-02-06 |   2
 postgres      | 2015-02-05 |   1
```

上記例では ALTER TABLE 文は各データベースで散発的に実行されており、頻度としては特に不自然ではないと思われます。

4.15.4. 監査対象の限定(参考情報)

postgresql.conf でのパラメータ設定はデータベースクラスタ全体に対して有効となるため、影響が大きすぎる場合があります。多くのパラメータはユーザ毎またはデータベース毎にカスタマイズすることもできるため、その機能を利用して限定した範囲で監査ログを取得することも考えられます。

(設定例) user1 ユーザでの接続時には、log_statement パラメータを'all'と認識させる

```
=# ALTER USER user1 SET log_statement='all';
ALTER ROLE

=# SELECT d.datname,
-#        r.rolname,
-#        s.setconfig
-# FROM   pg_db_role_setting s
-# LEFT   OUTER JOIN pg_database d ON (s.setdatabase = d.oid)
-# LEFT   OUTER JOIN pg_roles    r ON (s.setrole = r.oid);

 datnam  | rolname    |      setconfig
---------+------------+---------------------
         | user1      | {log_statement=all                  -- user1 ユーザ専有の設定
```

4.16. 定期的なセッション情報の分析（ログイン失敗回数が多い接続試行）

対応する PCI DSS 要件	10.2, 10.3

ログイン失敗回数が多い場合、不正な行為を意図している可能性があります。

4.16.1. 認証エラーおよびログの確認

「4.10.2. 認証エラーとログ出力例」をご参照下さい。

4.16.2. 認証エラーの検出

主として監視すべき対象としては認証エラーと考えられます。SQLSTATE の '28000'および'28P01'が該当します。
例） 認証エラーが発生した日時とユーザ名を取得
条件: SQLSTATE が”28000”または”28P01”
集計: message 別の合計値

```
=# SELECT message,
-#        count(*) AS count
-# FROM   postgres_log
-# WHERE  sql_state_code IN ('28000','28P01')
-# GROUP  BY message;
                                                                          message                                                                           | count
----------------------------------------------------------------------------------------------------------------------------------------------------------------+-------
password authentication failed for user ”user1”                                                                                                                 |   931
Peer authentication failed for user ”user2”                                                                                                                     |    20
no pg_hba.conf entry for host ”[local]”, user ”user1”, database ”database1”, SSL off                                                                            |     3
```

上位にランクされたエラーについては、対応を検討します。
上記例では、user1 ユーザに対するパスワード認証のエラーが突出しており、調査が必要です。

4.17. 定期的なセッション情報の分析（長時間に渡りログインしているセッション）

対応する PCI DSS 要件	10.2, 10.3

想定以上の長時間に渡るセッションは、不正なオペレーションを意図している可能性があります。

切断時のサーバログには、接続時間やユーザ、データベース、接続元などの過去のセッション情報が出力されます。それらからセッションの傾向を分析する事ができます。
なお現在のセッション情報はシステムカタログ pg_stat_activity から取得します。

4.17.1. 設定

サーバログに必要な情報を出力するために以下の設定を行います。

1. postgresql.conf の設定

表 4.22: サーバログに接続および切断情報を出力する設定例）

項目	概要
log_connections	'on'に設定。接続時の情報をサーバログに出力。デフォルトは'off'。
log_disconnections	'on'に設定。切断時の情報をサーバログに出力。デフォルトは'off'。

2. リロードによる即時反映

上記パラメータはリロードにより即時に反映させることができます。

(実行例）

```
$ pg_ctl reload
server signaled
```

パラメータに変更があった場合はサーバログに変更の旨が出力されます。

(サーバログ出力例）

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "log_connections" changed to "on"
LOG:  parameter "log_disconnections" changed to "on"
```

4.17.2. 接続／切断およびログの確認

接続および切断を実行し、サーバログに出力させます。

実行例）

```
$ psql -U user1 database1              -- 接続

psql (9.3.5)
"help" でヘルプを表示します.

=> ¥q                                  -- 切断
```

サーバログ出力例）　stderr 形式の場合、接続に 2 行、および切断時に 1 行が出力されています。
（認証方式により若干の違いはあります）

```
LOG:  connection received: host=[local]
LOG:  connection authorized: user=user1 database=database1
LOG:  disconnection: session time: 0:07:06.161 user=user1 database=database1 host=[local]
```

表 4.23: サーバログ出力内容詳細

レベル	出力内容
LOG	connection received: host=[local] 認証要求を受けた事を示しています(接続は完了していません)。 log_connections パラメータが on の場合に出力されます。
LOG	connection authorized: user=user1 database=database1 認証が成功し、接続が完了した事を示しています。 log_connections パラメータが on の場合に出力されます。
LOG	disconnection: session time: 0:07:06.161 user=user1 database=database1 host=[local] 切断した事を示しています。接続情報(接続していた時間、データベース、ユーザ)を出力します。 log_disconnections パラメータが on の場合に出力されます。

サーバログ出力例)　csvlog 形式の場合 (便宜上、空白行を挿入)

```
2015-02-05 19:24:24.074 JST,,,12089,"",54f82ed8.2f39,1,"",2015-02-05 19:24:24 JST,,0,
LOG,00000,"connection received: host=[local]",,,,,,,,,,""

2015-02-05 19:24:24.076 JST,"user1","database1",12089,"[local]",54f82ed8.2f39,2,
"authentication",2015-02-05 19:24:24 JST,1/16,0,LOG,00000,"connection authorized:
user=user1 database=database1",,,,,,,,,,""

2015-02-05 19:24:36.460 JST,"user1","database1",12089,"[local]",54f82ed8.2f39,3,"idle",
2015-02-05 19:24:24 JST,,0,LOG,00000,"disconnection: session time: 0:00:12.386 user=user1
database=database1 host=[local]",,,,,,,,,,"psql"
```

4.17.3. 接続情報の検出

postgres_log テーブルを参照します。postgres_log テーブルの内容については 4.6.2 項をご参照ください。

例)　切断時の情報から接続時間が 30 分以上経過したセッション情報を取得
条件: 切断時のデータであり　かつ接続時間が 30 分以上(いずれも message 列を参照)

```
=# SELECT log_time,
-#        user_name,
-#        database_name,
-#        connection_from,
-#        message,
-#        application_name,
-#        substr(message,30,7) AS sess_time
-# FROM   postgres_log
-# WHERE  message LIKE 'disconnection:%'
-# AND    substr(message,30,7) > '0:30:00';

-[ RECORD 1 ]----+-----------------------------------------------------------------
log_time         | 2015-01-26 18:59:20.022+09
user_name        | user1
database_name    | database1
connection_from  | [local]
message          | disconnection: session time: 3:57:16.871 user=user1
                   database=database1 host=[local]
application_name | psql
sess_time        | 3:57:16
```

上記例では、localhost から user1 ユーザにて database1 データベースに対して psql を使用して、2015 年 01 月 26 日の 18 時 59 分 20 秒から 4 時間弱の接続を行っています。
監査情報などから、どのような操作を行っていたのかを確認します。

4.18. 定期的な DB アクセス情報の分析（スロークエリの傾向分析）

対応する PCI DSS 要件	10.2, 10.3

想定外のスロークエリに対しては、情報漏洩を意図した不正な SQL の可能性があります。

SQL の実行時間の取得には、主に以下の方式があります。
方法 1. log_min_duration_statement パラメータ有効化によるサーバログへの出力
方法 2. auto_explain(contrib モジュール）有効化による実行計画、実行時間、I/O 時間の出力

ここでは方法 1 を説明します。
方法 2.は多機能ですが、一定の負荷がかかり常に有効化するものではないため、常時監査にはあまり向いていません。
特に調査が必要な場合にのみ実施します。

4.18.1. 設定

サーバログに必要な情報を出力するために以下の設定を行います。

1. postgresql.conf の設定

表 4.24: サーバログへのスロークエリ出力の設定例）

項目	概要
log_min_duration_statement	SQL 文の所要時間の暫定的な閾値として、'2s'(2 秒）を設定します。 この時間以上掛かった SQL を出力します。 全ての SQL 文の所要時間を出力するには、0 を設定します。

2. リロードによる即時反映
   上記パラメータはリロードにより即時に反映させることができます。
   (実行例）

```
$ pg_ctl reload
server signaled
```

   パラメータに変更があった場合はサーバログに変更の旨が出力されます。
   (サーバログ出力例）

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter ”log_min_duration_statement” changed to ”2s”
```

4.18.2. SQL 実行およびログの確認

時間のかかる SQL として pgbench_accounts テーブルの全件削除を実行します。
実行例）

```
=> delete from pgbench_accounts ;
DELETE 1000000
```

サーバログ出力例）　stderr 形式の場合

```
LOG:  statement: delete from pgbench_accounts ;
LOG:  duration: 42649.515 ms
```

表 4.25: サーバログ出力内容詳細

レベル	出力内容
LOG	LOG: delete from pgbench_accounts ; 発行された SQL が出力されます。SQL 発行時点に出力されます(SQL の完了を待ちません）。 log_statement による出力です。

LOG	LOG: duration: 42649.515 ms SQL の実行時間が出力されます。SQL が完了した時点で出力されます。 実行時間が log_min_duration_statement パラメータの閾値を超えた事による出力です。

(サーバログ出力例) csvlog 形式の場合 (便宜上、空白行を挿入)

```
2015-01-15 10:50:17.135 JST,"user1","database1",32677,"[local]",54b71cca.7fa5,3,"idle",
2015-01-15 10:50:02 JST,1/6,0,LOG,00000,"statement: delete from gbench_accounts ;
",,,,,,,,,"psql

2015-01-15 10:50:59.784 JST,"user1","database1",32677,"[local]",54b71cca.7fa5,4,"DELETE",
2015-01-15 10:50:02 JST,1/0,0,LOG,00000,"duration: 42649.515 ms",,,,,,,,,"psql"
```

4.18.3. スロークエリの検出

postgres_log テーブルを参照します。postgres_log テーブルの内容については 4.6.2 項をご参照ください。

取得例) スロークエリの取得
条件: message に文字列'duration:'が含まれる
結合: スロークエリの message には SQL が含まれていないため、対応する SQL 含まれているレコード(同一 process_id で message に 'statement:' が含まれているレコード) と自己結合

```
=# SELECT d.log_time,
-#        d.user_name,
-#        d.database_name,
-#        d.connection_from,
-#        s.message AS statement,
-#        d.message AS duration,
-#        d.application_name
-# FROM   (SELECT * FROM postgres_log WHERE message LIKE 'duration:%')  d,
-#        (SELECT * FROM postgres_log WHERE message LIKE 'statement:%') s   --- SQL 取得用
-# WHERE  s.process_id = d.process_id;

-[ RECORD 1 ]----+---------------------------------------------
log_time         | 2015-01-15 10:50:59.784+09
user_name        | pgbench
database_name    | pg93
connection_from  | [local]
statement        | statement: delete from pgbench.pgbench_accounts ;
duration         | duration: 42649.515 ms
application_name | psql
```

上記例では、pgbench_accounts テーブルの全件削除であることが分かります。バッチ処理によるものであり問題ありません。

4.19. 定期的な DB アクセス情報の分析（大量のリソースを消費する SQL の傾向分析）

対応する PCI DSS 要件	10.2, 10.3

想定外にリソースを消費する SQL はスロークエリと同様に、情報漏洩を意図した不正な SQL の可能性があります。
以下の手法は常時監視にはあまり向いていません。調査が必要な場合に一時的に設定します。

4.19.1. auto_explain

auto_explain を有効化する事で SQL 毎のディスク I/O 時間と所要時間を取得する事ができます。
PostgreSQL に標準で実装されている contrib モジュールの1つです。
PostgreSQL のマニュアル（F.3. auto_explain）にも説明がありますのでご参照ください。

本来は SQL のパフォーマンスを診断するツールですが、監査的な利用方法も考えられます。
例えば日中では軽微な SQL しか実行されないはずが、大量の DISK I/O を伴う SQL が発行された場合には、不正な SQL の可能性があります。

4.19.2. 設定

サーバログに必要な情報を出力するために以下の設定を行います。

1. auto_explain のコンパイル(必要な場合のみ)
   contrib モジュールはコンパイルする必要があります。PostgreSQL を RPM パッケージから導入していればコンパイル済みですが、ソースからコンパイルしている場合は、コンパイルされていない可能性があります。
   $PGHOME/lib/postgresql/auto_explain.so ライブラリがあれば導入されています(ディレクトリ構造は異なる場合があります)。ライブラリが存在しない場合は、以下のいずれかの方式でコンパイルします。
   データベースクラスタはすべて停止済みとします。

   A. contrib 全体のコンパイル

```
$ cd <ソースツリー>/contrib
$ make && make install
```

   B. contrib/auto_explain 単体のコンパイル

```
$ cd <ソースツリー>/contrib/auto_explain
$ make && make install
```

2. postgresql.conf の設定
   下記パラメータの設定にて取得する情報を増やすことができますが、負荷とのトレードオフになります。
   shared_preload_libraries パラメータ以外はリロードで反映可能であるため、動的に調整することができます。

表 4.26: auto_explain 関連の設定例

項目	概要
shared_preload_libraries	'auto_explain' を設定、または追記。反映には再起動が必要。
track_io_timing	'on'を設定し SQL 毎に DISK I/O 時間を取得。デフォルトは'off'。
auto_explain.log_min_duration	'2s'を設定。auto_explain の取得対象となる SQL の実行時間の閾値。 デフォルトは-1（取得しない）。
auto_explain.log_analyze	TRUE を設定。EXPLAIN ANALYZE 形式で出力。デフォルトは無効。
auto_explain.log_buffers	TRUE を設定。EXPLAIN (ANALYZE, BUFFERS) 形式で出力。 auto_explain.log_analyze =true である事が必要。デフォルトは無効。
auto_explain.log_timing	TRUE を設定。EXPLAIN (ANALYZE, TIMING off) 形式で出力。 デフォルトは有効。

3. 再起動による反映

auto_explain の有効化には再起動が必要です。

(実行例)

```
$ pg_ctl restart -m fast -w
waiting for server to shut down.... done
server stopped
waiting for server to start....
  ～中略～
 done
server started
$
```

4.19.3. SQL 実行及びサーバログの確認

時間のかかる SQL として pgbench_accounts テーブルの並び替えを行います。

(実行例)

```
 => SELECT * FROM pgbench_accounts ORDER BY aid,bid;
```

(サーバログ出力例) 実行時間 7.6 秒の内、DISK read 時間が 2.3 秒　(便宜上、空白行を挿入)

```
LOG:  statement: select * from pgbench_accounts order by aid,bid;
LOG:  duration: 7637.998 ms  plan:
        Query Text: SELECT * FROM pgbench_accounts ORDER BY aid,bid;
        Sort  (cost=284470.34..286970.34 rows=1000000 width=97) (actual rows=1000000
               loops=1)
          Sort Key: aid, bid
          Sort Method: external sort  Disk: 104552kB
          Buffers: shared read=15874, temp read=13069 written=13069
          I/O Timings: read=2260.124
          ->  Seq Scan on pgbench_accounts  (cost=0.00..25874.00 rows=1000000 width=97)
              (actual rows=1000000 loops=1)
                Buffers: shared read=15874
                I/O Timings: read=2260.124
```

表 4.27: サーバログ出力内容詳細

レベル	出力内容
LOG	LOG: statement: select * from pgbench_accounts order by aid,bid; 発行された SQL が出力されます。SQL 発行時点に出力されます(SQL の完了を待ちません)。 log_statement による出力です。
LOG	LOG: duration: 7637.998 ms plan: ～以下略～ 実行時間が auto_explain.log_min_duration パラメータの閾値を超えた事による出力です。 SQL の終了時点で出力されます。

(サーバログ出力例)　csvlog 形式の場合　(便宜上、空白行を挿入)

```
2015-01-26 15:04:41.406
JST,"user1","database1",8831,"[local]",54c5d85b.227f,29,"idle",2015-01-26 15:02:03
JST,1/16,0,LOG,00000,"statement: select * from pgbench_accounts order by
aid,bid;",,,,,,,,,"psql"

2015-01-26 15:04:49.045 JST,"user1","database1",8831,"[local]",54c5d85b.227f,30,"SELECT",
2015-01-26 15:02:03 JST,1/16,0,LOG,00000,"duration: 7637.998 ms  plan:
Query Text: select * from pgbench_accounts order by aid,bid;
Sort  (cost=284470.34..286970.34 rows=1000000 width=97) (actual rows=1000000 loops=1)
  Sort Key: aid, bid
  Sort Method: external sort  Disk: 104552kB
  Buffers: shared read=15874, temp read=13069 written=13069
  I/O Timings: read=226.124
  ->  Seq Scan on pgbench_accounts  (cost=0.00..25874.00 rows=1000000 width=97)
        (actual rows=1000000 loops=1)
        Buffers: shared read=15874
        I/O Timings: read=226.124",,,,,,,,,"psql"
```

4.19.4. 大量の DISK read が発生した SQL の抽出

postgres_log テーブルを参照します。postgres_log テーブルの内容については 4.6.2 項をご参照ください。

取得例) スロークエリの取得
条件: DISK read が 100 ミリ秒以上
メッセージに含まれている “duration:“と “I/O Timings:”の数値を小数点以下第一位までで抽出します。

```
=# SELECT message,
-#        substr(message,11,strpos(message,'.')-9) AS duration,
-#        substr(message2,
(#               strpos(message2,'I/O Timings: read=')+18,
(#               strpos(message2,'.')-17) AS io_timing
-# FROM   (SELECT message,
(#                substr(message,strpos(message,'I/O Timings: read=')) AS message2
(#         FROM   postgres_log) postgres_log2
-# WHERE  strpos(message,'I/O Timings: read=') > 0;

-[ RECORD 1 ]----------------------------------------------------------------------
message    | duration: 7637.998 ms  plan:
           | Query Text: select aid,bid,abalance,filler from pgbench_accounts
                         order by aid,bid;
           | Sort  (cost=284470.34..286970.34 rows=1000000 width=97) (actual rows=1000000
                  loops=1)
           |   Sort Key: aid, bid
           |   Sort Method: external sort  Disk: 104552kB
           |   Buffers: shared read=15874, temp read=13069 written=13069
           |   I/O Timings: read=226.124
           |   ->  Seq Scan on pgbench_accounts  (cost=0.00..25874.00 rows=1000000
                    width=97) (actual rows=1000000 loops=1)
           |         Buffers: shared read=15874
           |         I/O Timings: read=226.124
duration   | 7637.9                                  -- duration   (小数点第一位まで)
io_timing  | 226.1                                   -- I/O Timing (小数点第一位まで)
```

上記結果では、pgbench_accounts テーブルに対する全件の並び替えであり、外部ソートが行なわれているため、Disk read 時間が大きくなっていますが、バッチ処理であり問題ありません。

4.20. 定期的な DB アクセス情報の分析（エラーで終了している SQL の傾向分析）

対応する PCI DSS 要件	10.2, 10.3

エラーで終了している SQL では、権限のないテーブルへの不正なアクセスを試みている可能性があります。

4.20.1. 設定

サーバログに必要な情報を出力するために以下の設定を行います。

1. postgresql.conf の設定

表 4.28: サーバログへのエラー出力設定例

項目	概要
log_min_messages	warning（デフォルト）に設定。通常はデフォルトで問題ありません。 エラーのメッセージを出力するには、error 以下(error / warning / notice / info など)を設定します。

2. リロードによる即時反映

上記パラメータはリロードにより即時に反映させることができます。

(実行例)

```
$ pg_ctl reload
server signaled
```

パラメータに変更があった場合はサーバログに変更の旨が出力されます。

(サーバログ出力例)

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "log_min_messages" changed to "notice"
```

4.20.2. SQL 実行及びサーバログの確認

権限のないテーブルへのアクセスを試行します。

(実行例)　スキーマ schema1 への USAGE 権限のないユーザがアクセスを試みてエラーが発生

```
 => SELECT * FROM schema1.table1;
ERROR:  permission denied for schema schema1
行 1: SELECT * FROM schema1.table1;
                    ^
```

(サーバログ出力例) stderr の場合

```
LOG:  statement: SELECT * FROM schema1.table1;
ERROR:  permission denied for schema schema1 at character 15
STATEMENT:  SELECT * FROM schema1.table1;
```

表 4.29: サーバログ出力内容詳細

レベル	出力内容
LOG	LOG: statement: SELECT * FROM schema1.table1; 発行された SQL が出力されます。SQL 発行時点に出力されます(SQL の完了を待ちません)。 log_statement パラメータによる出力です。
ERROR	permission denied for schema schema1 at character 15 SQL 実行でエラーが発生した場合に、原因とエラー箇所が出力されます。 log_min_messages パラメータによる出力です。
-	STATEMENT: SELECT * FROM schema1.table1;

	SQL 実行でエラーが発生した場合に、SQL 文が出力されます。

サーバログ出力例） csvlog 形式の場合 （便宜上、空白行を挿入）

```
2015-02-05 16:24:28.568 JST,"user1","database1",25394,"[local]",54d31a9f.6332,3,"idle",
2015-02-05 16:24:15 JST,1/21,0,LOG,00000,"statement: SELECT * FROM schema1.table1;
",,,,,,,,,"psql"

2015-02-05 16:24:28.630 JST,"user1","database1",25394,"[local]",54d31a9f.6332,4,
"SELECT",2015-02-05 16:24:15 JST,1/21,0,ERROR,42501,"permission denied for schema
schema1",,,,,,"SELECT * FROM schema1.table1;",15,,"psql"
```

4.20.3. エラーが発生した SQL の抽出

postgres_log テーブルを参照します。postgres_log テーブルの内容については 4.6.2 項をご参照ください。

取得例) パーミッションエラーが発生した SQL の取得
条件 ：エラーが発生し、かつエラー原因がパーミッション拒否によるもの

```
=# SELECT log_time,
-#        user_name,
-#        database_name,
-#        connection_from,
-#        query,
-#        message,
-#        sql_state_code,
-#        application_name
-# FROM   postgres_log
-# WHERE  error_severity ='ERROR'                      --- エラーが発生
-# AND    message LIKE 'permisson denied for %';       --- 原因はパーミッション拒否

-[ RECORD 1 ]----+-------------------------------------------------------------------
log_time         | 2015-02-05 16:24:28.63+09
user_name        | user1
database_name    | database1
connection_from  | [local]
query            | SELECT * FROM schema1.table1;
message          | permission denied for schema schema1
sql_state_code   | 42501
application_name | psql
```

実行ユーザ、日付、クライアント、アプリケーション名などから、情報漏洩につながるものかどうかを判断します。
上記例では、schema1 スキーマの table1 テーブルに対して、パーミッション拒否のエラーが発生しています。
アプリケーションの設計上、user1 ユーザは schema1 スキーマへアクセスする必要はないのであれば、別途意図を確認します。

4.21. 定期的な DB アクセス情報の分析（全件取得の傾向分析）

対応する PCI DSS 要件	10.2, 10.3

アプリケーションからの正規の SQL では無条件の検索が行なわれないはずのテーブルに対して、無条件の検索 (SELECT / COPY TO)が行なわれている場合は不正な SQL の可能性があります。

4.21.1. 設定

サーバログに必要な情報を出力するために以下の設定を行います。

1. postgresql.conf の設定

表 4.30: サーバログへの SQL 出力設定例

項目	概要
log_statement	全ての SQL を取得するため、'all'に設定 node : 取得しない ddl : DDL 文(CREATE / ALTER /DROP など)を取得 mod : 全ての DDL 文および更新 (UPDATE/DELETE/INSERT/TRUNCATE/COPY FROM)を取得 all : 全ての SQL 文を取得

2. リロードによる即時反映

上記パラメータはリロードにより即時に反映させることができます。

(実行例)

```
$ pg_ctl reload
server signaled
```

パラメータに変更があった場合はサーバログに変更の旨が出力されます。

(サーバログ出力例)

```
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "log_min_messages" changed to "notice"
```

4.21.2. SQL 実行及びサーバログの確認

pgbench_accounts テーブルからの全件取得

(実行例)

```
=> SELECT * pgbench_accounts;
```

サーバログ出力例)

```
LOG:  statement: SELECT * FROM pgbench_accounts;
```

表 4.31: サーバログ出力内容詳細

レベル	出力内容
LOG	LOG: statement: select * from pgbench_accounts 発行された SQL が出力されます。SQL 発行時点に出力されます(SQL の完了を待ちません)。 log_statement による出力です。

(サーバログ出力例)　csvlog 形式の場合　(便宜上、空白行を挿入)

```
015-02-05 19:18:07.296 JST,"bench","pg93",26390,"[local]",54d3435c.6716,3,"idle",
2015-02-05 19:18:04 JST,1/6,0,LOG,00000,"statement: SELECT * FROM pgbench_accounts;"
,,,,,,,,,"psql"
```

4.21.3. 無条件 SELECT 文の抽出

postgres_log テーブルを参照します。postgres_log テーブルの内容については 4.6.2 項をご参照ください。

取得例) 無条件の SELECT 文
条件　: SELECT 文である、 WHERE 句が無い、成功している

```
=# SELECT log_time,
-#        user_name,
-#        database_name,
-#        connection_from,
-#        message,
-#        application_name
-# FROM   postgres_log
-# WHERE  upper(message) LIKE 'STATEMENT: SELECT%'     --- SELECT 文
-# AND    upper(message) NOT LIKE '%WHERE%'            --- WHERE 句がない
-# AND    sql_state_code = '00000';                    --- 成功している

-[ RECORD 1 ]----+-----------------------------------------------------------------
log_time         | 2015-01-26 15:04:41.406+09
user_name        | user1
database_name    | database1
connection_from  | [local]
message          | statement: select aid,bid,abalance,filler from pgbench_accounts
                            order by aid,bid;
application_name | psql
```

SQL、実行時間、ユーザ、データベース、アプリケーションなどから通常業務によるものかどうかを確認します。
上記例では、pgbench_accounts テーブルへの全件検索です。バッチ処理によるものであり、問題ありません。

4.22. 行単位のアクセス制御

対応する PCI DSS 要件	6.3.1.5

PostgreSQL 9.5 の新機能として開発面で期待の大きい「Row Level Security（RLS）」があります。テーブルにポリシーを設定することで、テーブルに対して問い合わせが行われた際に、同一の SQL であっても処理を実行したユーザに応じて表示する行データを制御する機能です。行単位でのアクセス制御機能が標準で組み込まれることは PostgreSQL のセキュリティ機能強化の大きな一歩と言えます。

なおスーパーユーザは RLS ポリシーをバイパスするため無効となります。

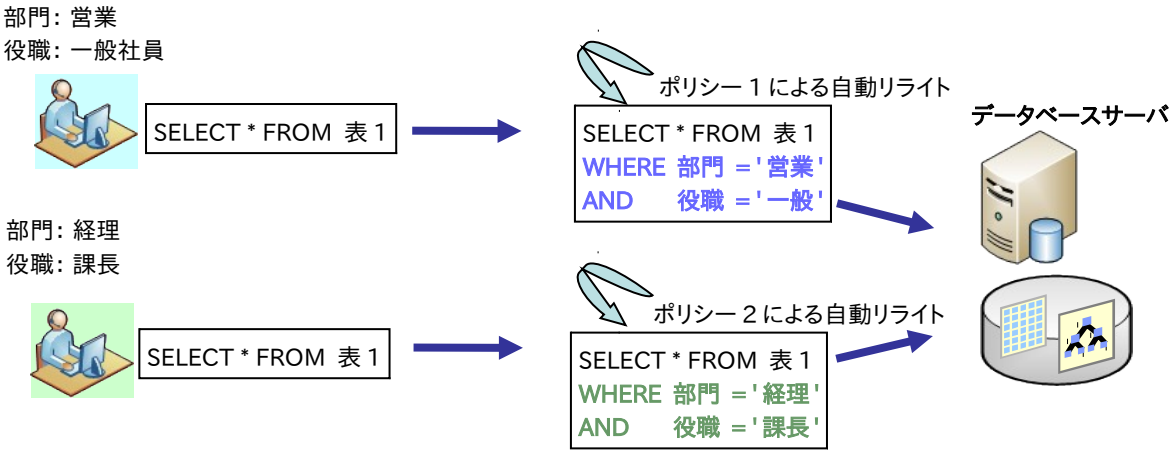


図 4.3: RLS の概念図

4.22.1. 設定

サーバログに必要な情報を出力するために以下の設定および確認を行います。

1. テーブルに対して RLS を有効化
2. RLS ポリシーの作成
3. row_security パラメータの確認
4. 対象ユーザの bypassrls 属性の確認

1. テーブルに対して RLS を有効化

テーブルに対して 1 回設定します。

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    DISABLE ROW LEVEL SECURITY              --- RLS 無効化
    ENABLE ROW LEVEL SECURITY               --- RLS 有効化
    FORCE ROW LEVEL SECURITY                --- RLS 有効化(オーナーに対しても有効)
    NO FORCE ROW LEVEL SECURITY             --- RLS 有効化(オーナーに対しては無効)
```

(実行例)emp テーブルに対して RLS を有効化

```
■設定
=# ALTER TABLE emp ENABLE ROW LEVEL SECURITY;
ALTER TABLE

■確認
=# SELECT relrowsecurity
-# FROM   pg_class
-# WHERE  relname = 'emp';
 relrowsecurity
----------------
 t
```

```
(1 行)
```

2. RLS ポリシーの作成
RLS ポリシーを作成します。テーブルに対して複数作成できます。

```
CREATE POLICY name ON table_name
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

(RLS ポリシー作成例)
対象テーブル: emp テーブル
対象操作　　: 全ての操作(SELECT/INSERT/UPDATE/DELETE)
対象ユーザ　: user_a ユーザ
条件　　: deptno=30

```
■作成
=# CREATE POLICY emp_user_a ON emp FOR ALL
-# TO user_a
-# USING (deptno=30);
CREATE POLICY

■確認
=# ¥d emp
    テーブル "public.emp"
    列    |   型    | 修飾語
----------+---------+--------
 empno    | numeric |
 ename    | text    |
 job      | text    |
 mgr      | numeric |
 hiredate | date    |
 sal      | numeric |
 comm     | numeric |
 deptno   | numeric |
Policies:
    POLICY "emp_user_a" FOR ALL
      TO user_a
      USING ((deptno = (30)::numeric))
```

なお同一テーブルおよび同一ユーザにて複数のポリシーを設定する事も可能です。その場合は or 条件で評価されるため、どちらかのポリシーに合致しているデータが対象となります。

3. row_security パラメータの確認
row_security パラメータは行セキュリティポリシーの適用によってエラーを生じさせるかどうかを制御します。
on(デフォルト)である事を確認します。セッション単位での変更も可能です。
on ：　通常通りポリシーが適用されます。デフォルト。
off ：　少なくともひとつのポリシーが適用されたクエリは失敗します。

(確認例) row_security パラメータが on である事を確認

```
=# show row_security ;
 row_security
--------------
 on
(1 行)
```

4. 対象ユーザの bypassrls 属性の確認
対象ユーザに bypassrls 属性にて RLS ポリシーをバイパスするかどうかを制御できます。
設定されていない(デフォルト)事を確認します。

(確認例) user_a ユーザに BYpass RLS 属性が設定されていない事を確認

BYpass RLS 属性が設定されている場合、pg95 ユーザのように表示されます。
またスーパーユーザは BYpass RLS 属性が設定されていなくても RLS をバイパスします。

```
=# ¥du
                                   ロール一覧
 ロール名 |                                 属性                                 | メンバー
----------+------------------------------------------------------------------------+----------
 pg95     | Bypass RLS                                                             | {}        --- 設定あり
 postgres | スーパーユーザ, ロールを作成できる, DB を作成できる                    | {}
 user_a   |                                                                        | {}        --- 設定なし
 user_b   |                                                                        | {}        --- 設定なし
```

4.22.2. 動作確認

1. SELECT の動作確認

emp テーブルには、次のように deptno=10,20,30 の行が存在します。

```
=# SELECT * FROM emp;
 empno | ename  |    job    | mgr  |  hiredate  | sal  | comm | deptno
-------+--------+-----------+------+------------+------+------+--------
  7369 | SMITH  | CLERK     | 7902 | 1980-12-17 |  800 |      |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 1981-02-20 | 1600 |  300 |     30
  7521 | WARD   | SALESMAN  | 7698 | 1981-02-22 | 1250 |  500 |     30
  7566 | JONES  | MANAGER   | 7839 | 1981-04-02 | 2975 |      |     20
  7654 | MARTIN | SALESMAN  | 7698 | 1981-09-28 | 1250 | 1400 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 1981-05-01 | 2850 |      |     30
  7782 | CLARK  | MANAGER   | 7839 | 1981-06-09 | 2450 |      |     10
  7788 | SCOTT  | ANALYST   | 7566 | 1982-12-09 | 3000 |      |     20
  7839 | KING   | PRESIDENT |      | 1981-11-17 | 5000 |      |     10
  7844 | TURNER | SALESMAN  | 7698 | 1981-09-08 | 1500 |    0 |     30
  7876 | ADAMS  | CLERK     | 7788 | 1983-01-12 | 1100 |      |     20
  7900 | JAMES  | CLERK     | 7698 | 1981-12-03 |  950 |      |     30
  7902 | FORD   | ANALYST   | 7566 | 1981-12-03 | 3000 |      |     20
  7934 | MILLER | CLERK     | 7782 | 1982-01-23 | 1300 |      |     10
  8001 | O'Neal | CLERK     | 7782 | 1982-02-27 | 1500 |      |     10
(15 行)
```

user_a ユーザにて emp テーブルに SELECT すると、自動的に条件に deptno=30 が付与されます。

```
=> ¥connect postgres user_a;
データベース "postgres" にユーザ"user_a"として接続しました。

=> SELECT * FROM emp;
 empno | ename  |   job    | mgr  |  hiredate  | sal  | comm | deptno
-------+--------+----------+------+------------+------+------+--------
  7499 | ALLEN  | SALESMAN | 7698 | 1981-02-20 | 1600 |  300 |     30
  7521 | WARD   | SALESMAN | 7698 | 1981-02-22 | 1250 |  500 |     30
  7654 | MARTIN | SALESMAN | 7698 | 1981-09-28 | 1250 | 1400 |     30
  7698 | BLAKE  | MANAGER  | 7839 | 1981-05-01 | 2850 |      |     30
  7844 | TURNER | SALESMAN | 7698 | 1981-09-08 | 1500 |    0 |     30
  7900 | JAMES  | CLERK    | 7698 | 1981-12-03 |  950 |      |     30
(6 行)

=> EXPLAIN SELECT * FROM emp;
                       QUERY PLAN
--------------------------------------------------------
 Seq Scan on emp  (cost=0.00..13.88 rows=2 width=228)
   Filter: (deptno = '30'::numeric)
(2 行)
```

条件 deptno=10 を付与すると RLS ポリシーと条件が競合し、0件となります。

```
=> SELECT * FROM emp WHERE deptno=10;
 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 行)

=> EXPLAIN SELECT * FROM emp WHERE deptno=10;
                            QUERY PLAN
------------------------------------------------------------------
 Subquery Scan on emp  (cost=0.00..13.90 rows=1 width=228)
```

```
   Filter: (emp.deptno = '10'::numeric)
   ->  Seq Scan on emp emp_1  (cost=0.00..13.88 rows=2 width=228)
         Filter: (deptno = '30'::numeric)
(4 行)
```

RLS ポリシー設定のないユーザでは 0 件となります。
以下は RLS ポリシー設定のない user_b での操作です。

```
=> ¥connect postgres user_b;
データベース "postgres" にユーザ"user_b"として接続しました。

=> SELECT * FROM emp;
 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 行)

=> EXPLAIN SELECT * FROM emp;
                QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=0)
   One-Time Filter: false
(2 行)
```

2. INSERT の動作確認
deptno=30 のデータは INSERT できます。
それ以外(以下の例では 10)を指定するとエラーが発生します。
なお INSERT の場合、実行計画には RLS ポリシーの影響はありません。

```
=> ¥connect postgres user_a;
データベース "postgres" にユーザ"user_a"として接続しました。

=> INSERT INTO emp (empno,ename,deptno) VALUES (9000,'RLS TEST',30);
INSERT 0 1

=> INSERT INTO emp (empno,ename,deptno) VALUES (9000,'RLS TEST',10);
ERROR:  new row violates row-level security policy for table "emp"

=> EXPLAIN INSERT INTO emp (empno,ename,deptno) VALUES (9000,'RLS TEST',10);
                    QUERY PLAN
--------------------------------------------------
 Insert on emp  (cost=0.00..0.01 rows=1 width=0)
   ->  Result  (cost=0.00..0.01 rows=1 width=0)
(2 行)
```

RLS ポリシー設定のないユーザではエラーとなります。
以下は RLS ポリシー設定のない user_b での操作です。

```
=> ¥connect postgres user_b;
データベース "postgres" にユーザ"user_b"として接続しました。

=> INSERT INTO emp (empno,ename,deptno) VALUES (9000,'RLS TEST',30);
ERROR:  new row violates row-level security policy for table "emp"
```

3. UPDATE の動作確認
deptno=30 のデータは UPDATE できます。
それ以外(以下の例では 10)を指定すると RLS ポリシーの条件と競合して0件となります。

```
=> ¥connect postgres user_a;
データベース "postgres" にユーザ"user_a"として接続しました。

=> UPDATE emp SET sal=sal*1.1 WHERE deptno=30;
UPDATE 6

=> UPDATE emp SET sal=sal*1.1 WHERE deptno=10;
UPDATE 0

=> EXPLAIN UPDATE emp SET sal=sal*1.1 WHERE deptno=10;
                                  QUERY PLAN
------------------------------------------------------------------------------
```

```
 Update on emp emp_1  (cost=0.00..13.92 rows=1 width=234)
   ->  Subquery Scan on emp  (cost=0.00..13.92 rows=1 width=234)
         Filter: (emp.deptno = '10'::numeric)
         ->  LockRows  (cost=0.00..13.89 rows=2 width=234)
               ->  Seq Scan on emp emp_2  (cost=0.00..13.88 rows=2 width=234)
                     Filter: (deptno = '30'::numeric)
```

deptno=30 のデータをそれ以外(以下の例では 10)への更新を試みるとエラーが発生します。

```
=> UPDATE emp SET deptno=10 WHERE deptno=30;
ERROR:  new row violates row-level security policy for table "emp"
```

RLS ポリシー設定のないユーザでは 0 件となります。
以下は RLS ポリシー設定のない user_b での操作です。

```
=> ¥connect postgres user_b;
データベース "postgres" にユーザ"user_b"として接続しました。

=> UPDATE emp SET sal=sal*1.1 WHERE deptno=30;
UPDATE 0

=> EXPLAIN UPDATE emp SET sal=sal*1.1 WHERE deptno=30;
                    QUERY PLAN
--------------------------------------------------
 Update on emp  (cost=0.00..0.01 rows=1 width=0)
   ->  Result  (cost=0.00..0.01 rows=1 width=0)
         One-Time Filter: false
(3 行)
```

4. DELETE の動作確認

deptno=30 のデータは DELETE できますが、それ以外(以下の例では 10)を指定すると 0 件となります。

```
=> ¥connect postgres user_a;
データベース "postgres" にユーザ"user_a"として接続しました。

=>  DELETE FROM emp WHERE deptno=30;
DELETE 7

=>  DELETE FROM emp WHERE deptno=10;
DELETE 0
```

RLS ポリシー設定のないユーザでは 0 件となります。
以下は RLS ポリシー設定のない user_b での操作です。

```
=> ¥connect postgres user_b;
データベース "postgres" にユーザ"user_b"として接続しました。

=>  DELETE FROM emp;
DELETE 0

=> EXPLAIN DELETE FROM emp;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Delete on emp emp_1  (cost=0.00..13.12 rows=1 width=6)
   ->  Subquery Scan on emp  (cost=0.00..13.12 rows=1 width=6)
         ->  LockRows  (cost=0.00..13.11 rows=1 width=6)
               ->  Result  (cost=0.00..13.10 rows=1 width=6)
                     One-Time Filter: false
                     ->  Seq Scan on emp emp_2  (cost=0.00..13.10 rows=1 width=6)
(6 行)
```

まとめ
以下では、ポリシーに合致するデータ(本例では deptno=30)をポリシーデータ、それ以外を非ポリシーデータと表記します。

表 4.32: RLS 挙動のまとめ

操作	RLS ポリシーユーザ ポリシーデータに対する処理	RLS ポリシーユーザ 非ポリシーデータに対する処理	非 RLS ポリシーユーザ
SELECT	ポリシー内で正常に SELECT 可能	正常終了(0件)	正常終了(0件)
INSERT	ポリシー内で正常に INSERT 可能	**エラー発生**	**エラー発生**
UPDATE	ポリシー内で正常に UPDATE 可能 **ただし非ポリシーデータへの更新となる処理ではエラー発生**	正常終了(0件)	正常終了(0件)
DELETE	ポリシー内で正常に DELETE 可能	正常終了(0件)	正常終了(0件)

4.22.3. 注意点

1. PostgreSQL 9.5 の新機能であるため、既存システムで本機能を使用するためにはメジャーバージョンアップが必要となります。

2. VIEW での対応と比較すると、テーブルレベルでの設定は抜け漏れを防ぐ事ができるというメリットがありますが、暗黙的に変換されるため設定が分かり辛くなるというリスクもあります。

3. テーブルに対して RLS ポリシーを有効化すると、ポリシーが作成されていないユーザは全データを参照できなくなります。RLS 制御の必要がないユーザには BYPASSRLS 属性を設定します。

(設定例)

```
■設定前（user_b ユーザから emp テーブルの全データが参照できない）

=> ¥connect postgres user_b;
データベース "postgres" にユーザ"user_b"として接続しました。

=> SELECT * FROM emp;
 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 行)

■設定　（user_b に BYPASSRLS 属性を設定）

=# ALTER ROLE user_b BYPASSRLS;
ALTER ROLE

=# ¥du
                                   ロール一覧
 ロール名 |                            属性                             | メンバー
----------+-----------------------------------------------------------+----------
 pg95     | Bypass RLS                                                | {}
 postgres | スーパーユーザ, ロールを作成できる, DB を作成できる | {}
 user_a   |                                                           | {}
 user_b   | Bypass RLS                                                | {}

■設定後　（user_b ユーザから emp テーブルの全データが参照できる）

=> ¥connect postgres user_b;
データベース "postgres" にユーザ"user_b"として接続しました。

=> SELECT * FROM emp;
 empno |  ename   |    job    | mgr  |  hiredate  |  sal   | comm | deptno
-------+----------+-----------+------+------------+--------+------+--------
  7369 | SMITH    | CLERK     | 7902 | 1980-12-17 |    800 |      |     20
  7566 | JONES    | MANAGER   | 7839 | 1981-04-02 |   2975 |      |     20
  7782 | CLARK    | MANAGER   | 7839 | 1981-06-09 |   2450 |      |     10
```

～以後略～

実行計画では RLS ポリシーを BYPASS している事は分からない

```
=> EXPLAIN SELECT * FROM emp;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on emp  (cost=0.00..13.10 rows=310 width=228)
(1 行)
```

4.23. SQL レベルのファイアウォール制御

対応する PCI DSS 要件	6.3.1.5

sql_firewall はアップタイム・テクノロジーズ合同会社 永安様作成の拡張モジュールです。[22]
PostgreSQL 上で実行可能な SQL を制限する事ができます。それにより SQL インジェクションを防ぐことを可能にしています。

以下の動作モードがあります。一般的には、学習モードから警告モードを経て、最終的に防御モードにて運用します。

表 4.33: sql_firewall モード一覧

モード	説明
学習モード （learning）	本モードでは、実行可能な SQL 文を学習させます。 sql_firewall を学習モードに設定することで、試験期間などにアプリケーションから発行される SQL 文を自動的に登録させる事ができます。
警告モード （permissive）	本モードでは、学習させた SQL 以外が実行された場合、警告が発生します（実行はできます）。 監査として使用する事ができます。
防御モード （enforcing）	本モードでは、学習させた SQL 以外が実行された場合、エラーが発生します。
無効モード (disabled)	本モードでは、sql_firewall を無効化します。

なお現時点(2016/02)での対応バージョンは 9.4 のみです。

4.23.1. 導入

以下の手順となります。

1. tar ボールのダウンロードおよびサーバへのアップ、tar ボールの解凍
2. コンパイル、リンク
3. sql_firewall エクステンションの作成
4. パラメータ設定および再起動

実行例を記載します。PostgreSQL 9.4 および sql_firewall 0.8.1 を使用しています。

1. tar ボールのダウンロードおよびサーバへのアップ、tar ボールの解凍
   Github で公開[23]されている tar ボールをダウンロードし、導入対象の PostgreSQL へアップロードします。

（解凍例）

```
$ unzip sql_firewall-develop.zip
Archive:  sql_firewall-develop.zip
39fdec885d748975e7475e05156b91eddff9d7a7
   creating: sql_firewall-develop/
  inflating: sql_firewall-develop/COPYRIGHT
  inflating: sql_firewall-develop/ChangeLog
  inflating: sql_firewall-develop/Makefile
  inflating: sql_firewall-develop/README.sql_firewall
  inflating: sql_firewall-develop/sql_firewall--0.8.sql
  inflating: sql_firewall-develop/sql_firewall.c
  inflating: sql_firewall-develop/sql_firewall.control
```

22 http://pgsqldeepdive.blogspot.jp/2015/08/postgresql-sql-firewall.html
23 https://github.com/uptimejp/sql_firewall1

2. コンパイル、リンク
   USE_PGXS=1 オプションを付与して make ,make install します。

（実行例）

```
$ cd sql_firewall-develop

$ make USE_PGXS=1
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-
labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -fwrapv -O2 -fpic
-I. -I./ -I/home/p941/p941_home/include/postgresql/server
-I/home/p941/p941_home/include/postgresql/internal -D_GNU_SOURCE -I/usr/include/libxml2
-c -o sql_firewall.o sql_firewall.c
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-
labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -fwrapv -O2 -fpic
-shared -o sql_firewall.so sql_firewall.o -L/home/p941/p941_home/lib -Wl,--as-needed -Wl,-
rpath,'/home/p941/p941_home/lib',--enable-new-dtags

$ make USE_PGXS=1 install
/bin/mkdir -p '/home/p941/p941_home/lib/postgresql'
/bin/mkdir -p '/home/p941/p941_home/share/postgresql/extension'
/bin/mkdir -p '/home/p941/p941_home/share/postgresql/extension'
/usr/bin/install -c -m 755  sql_firewall.so
'/home/p941/p941_home/lib/postgresql/sql_firewall.so'
/usr/bin/install -c -m 644 sql_firewall.control
'/home/p941/p941_home/share/postgresql/extension/'
/usr/bin/install -c -m 644 sql_firewall--0.8.sql
'/home/p941/p941_home/share/postgresql/extension/'
$
```

3. sql_firewall エクステンションの作成
   PostgreSQL を起動し、sql_firewall エクステンションを作成します。

（作成例）

```
$ pg_ctl start -w

$ psql

=# CREATE EXTENSION sql_firewall ;
CREATE EXTENSION

=# ¥dx
                                          インストール済みの拡張の一覧
        名前         | バージョン |  スキーマ  |                                説明
---------------------+------------+------------
+-------------------------------------------------------------------
 sql_firewall        | 0.8        | public     | Prevent query execution which is not
allowd by the rules
  ～以下略～
```

4. パラメータ設定および再起動
   postgresql.conf に以下を設定変更および追加します。
   sql_firewall.firewall パラメータにて sql_firewall のモードを設定します。
   ここでは sql_firewall のモードは学習モードの設定です。

```
shared_preload_libraries = 'sql_firewall'               --- sql_firewall のライブラリ
sql_firewall.firewall = 'learning'                      --- sql_firewall のモード設定
```

PostgreSQL の再起動で反映し、確認します。

```
$ pg_ctl restart -m fast -w

$ psql

=# show shared_preload_libraries ;
 shared_preload_libraries
```

```
-------------------------
 sql_firewall

=# show sql_firewall.firewall;
 sql_firewall.firewall
-----------------------
 learning
(1 行)
```

4.23.2. 学習モード

学習モードに設定した状態で、各種 SQL を実行します。
バインド変数や大文字小文字を織り交ぜます。

（実行例）

```
=> SELECT * FROM emp WHERE deptno=10;         --- 大文字/小文字
=> SELECT * FROM EMP WHERE DEPTNO=20;         --- 大文字
=> select * from emp where deptno=30;         --- 小文字
```

sql_firewall に登録されている SQL を確認します。
最初に実行された文字列で登録されます。
バインド変数や大文字/小文字の相違は吸収され、SQL の情報として保存されます。
（確認例）

```
=# SELECT query
-# FROM sql_firewall.sql_firewall_statements
-# WHERE query like '% emp%';
                            query                             | calls
--------------------------------------------------------------+-------
 SELECT * FROM emp WHERE deptno=?;                            |     3
```

4.23.3. 警告モード

sql_firewall を警告モードで動作します。
sql_firewall.firewall = 'permissive' と設定し再起動します。

```
sql_firewall.firewall = 'permissive'                    --- sql_firewall のモード設定
```

PostgreSQL の再起動で反映し、確認します。

```
$ pg_ctl restart -m fast -w

$ psql

=# show sql_firewall.firewall;
 sql_firewall.firewall
-----------------------
 permissive
```

学習済み SQL および未学習 SQL をそれぞれ実行して後者では警告が発生する事を認します。
（実行例）

```
■学習済み SQL

=# SELECT * FROM emp WHERE deptno=10;
 empno | ename  |    job    | mgr  |  hiredate  | sal  | comm | deptno
-------+--------+-----------+------+------------+------+------+--------
  7782 | CLARK  | MANAGER   | 7839 | 1981-06-09 | 2450 |      |     10
  7839 | KING   | PRESIDENT |      | 1981-11-17 | 5000 |      |     10
  ～以後略～

■未学習 SQL

=# SELECT * FROM dept WHERE deptno=10;
WARNING:  Prohibited SQL statement
WARNING:  Prohibited SQL statement
```

```
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
```

監査(log_statement=all)を設定している場合、サーバログ(csvlog)には以下の出力があります(深刻度以降の項目を抜粋)。

```
LOG,00000,"statement: SELECT * FROM emp WHERE deptno=10;",,,,,,,,"psql"
LOG,00000,"statement: SELECT * FROM dept WHERE deptno=10;",,,,,,,,,"psql"
WARNING,01000,"Prohibited SQL statement",,,,,,,,,"psql"
WARNING,01000,"Prohibited SQL statement",,,,,,,,,"psql"
```

4.23.4. 防御モード

sql_firewall を防御モードで動作します。
sql_firewall.firewall = 'enforcing' と設定し再起動します。

```
sql_firewall.firewall = 'enforcing'                    --- sql_firewall のモード設定
```

PostgreSQL の再起動で反映し、確認します。

```
$ pg_ctl restart -m fast -w

$ psql

=# show sql_firewall.firewall;
 sql_firewall.firewall
-----------------------
 enforcing
```

学習済み SQL および未学習 SQL をそれぞれ実行して後者ではエラーが発生する事を確認します。
(実行例)

```
■学習済み SQL

=# SELECT * FROM emp WHERE deptno=10;
 empno | ename  |    job    | mgr  |  hiredate  | sal  | comm | deptno
-------+--------+-----------+------+------------+------+------+--------
  7782 | CLARK  | MANAGER   | 7839 | 1981-06-09 | 2450 |      |     10
  7839 | KING   | PRESIDENT |      | 1981-11-17 | 5000 |      |     10
  ～以後略～

■未学習 SQL

=# SELECT * FROM dept WHERE deptno=10;
ERROR:  Prohibited SQL statement
```

監査(log_statement=all)を設定している場合、サーバログ(csvlog)には以下の出力があります(深刻度以降の項目を抜粋)。

```
LOG,00000,"statement: SELECT * FROM dept WHERE deptno=10;",,,,,,,,,"psql"
ERROR,2F003,"Prohibited SQL statement",,,,,,"SELECT * FROM dept WHERE deptno=10;",,,"psql"
```

4.23.5. 注意点

1. 学習モードでは、アプリケーションが発行するSQLだけでなく、メンテナンスに使用するSQLも学習させる必要があります。

   （失敗例）psql にてデータベース一覧を表示するコマンド(¥l)も未学習だと防御モードでエラー発生

```
=# ¥l
ERROR:  Prohibited SQL statement
```

2. 本ツールは防御モードに設定する事で、SQL インジェクションを防止するのが最大の目的です。ただし未学習の SQL は全て不正 SQL と見做されエラーとなるため、実運用にはややリスクがあります。防御モードで運用する前に学習モードおよび警告モードでの十分なテストが必要です。また警告モードのまま監査的な運用に留めるのも有力です。

3. 警告モードでの運用にて、警告メッセージは client_min_messages パラメータの調整で抑制できます。
   (実行例)

```
=# SELECT * FROM dept WHERE deptno=10;       --- 未学習 SQL にて警告メッセージ表示
WARNING:  Prohibited SQL statement
WARNING:  Prohibited SQL statement
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK

=# set client_min_messages = error;          --- client_min_messages パラメータの調整
SET

=# SELECT * FROM dept WHERE deptno=10;       --- 警告メッセージが抑制された
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
```

4. sql_firewall のモードの変更には再起動が必要です。

5. sql_firewall の設定はデータベースクラスタ全体に及びます。sql_firewall エクステンションを作成していないデータベースにも影響します。なお sql_firewall.sql_firewall_statements ビューが参照できるのは sql_firewall エクステンションを作成したデータベースだけです。

4.24. 不正アクセスのメール通知

対応する PCI DSS 要件	10.6

不正アクセスに対して、メール通知を行うことは、PostgreSQL の標準機能で実現することはできません。そのため、別の枠組みを利用する必要があります。

4.24.1. 実現手段

PostgreSQL の標準機能に加えて、ログを監視／収集する外部ツール、メール送付を行う外部ツールを組み合わせることで、本要件を満たすことができます。以下のような構成をとり、各ツールに適切なものを利用してください。

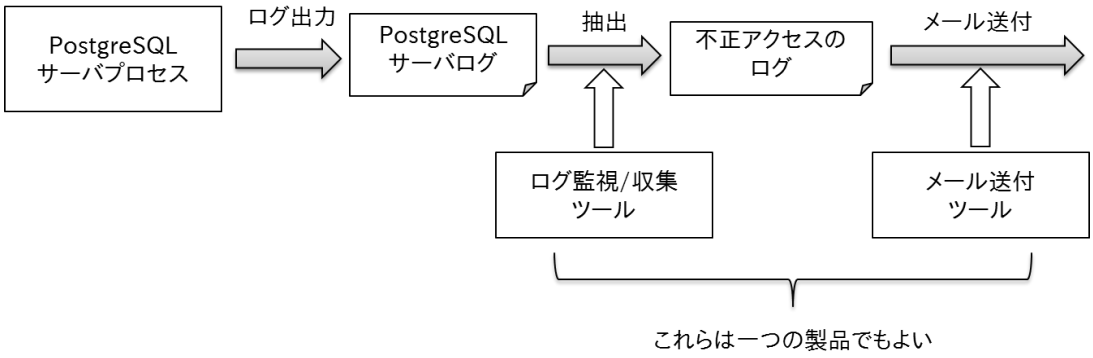


図 4.4: 不正アクセスのメール通知の仕組み

4.24.2. 適用例

ログ監視/収集ツールの代表的なものとして以下のものが挙げられます。

- logwatch[24]
- Zabbix[25]
- Hinemos[26]
  など

ここではログ監視/収集ツールとして、「logwatch」を利用する例を紹介します。他の外部ツールの導入手順・利用方法については、各ツールのドキュメントや関連サイトをお読みください。

logwatch は、サーバログを、レポート形式にまとめて毎日メールで通知するサーバログ監視ツールです。今回は、logwatch に、dalibo 社が公開している PostgreSQL 用のアタッチメント(以下、pgsql_logwatch[27]と呼ぶ)を利用して、PostgreSQL のサーバログのエラー通知をレポート形式にまとめます。

例として、前日の PostgreSQL サーバログのうち、ERROR 以上のものを、翌日午前 4 時に管理者に送付する例を紹介します。ここでは、サーバログをファイルに書き出すようにし、cron によって午前 4 時に logwatch が起動し、そのサーバログを調べてレポートを作成、管理者にメール送付するとします。

まず、postgresql.conf を以下のように指定します。ここで、pgsql_logwatch が正しく処理できるようにするために、

24 http://www.logwatch.org/
25 http://www.zabbix.jp/
26 http://www.hinemos.info/
27 https://github.com/dalibo/pgsql_logwatch

log_line_prefix を'%t 'で始めるようにします。

$PGDATA/postgresql.conf

```
# (中略)
# 接続情報をログに出力、ログの接頭を設定
log_connections = on                # 接続をログに出力する(反映には再起動が必要)
log_line_prefix = '%t [%u][%d][%r]'  # "日付時刻 [ユーザ名][データベース名][ホスト名]"

# ログをファイルに書き出すように設定
log_destination = 'stderr'                          # 標準エラー出力に表示(反映には再起動が必要)
logging_collerctor = on                             # 標準エラー出力をログファイルに書き出す
log_directory = 'pg_log'                            # ログを書き出すディレクトリを指定
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'  # ファイル名のフォーマットを指定
log_rotation_age = 1d                               # 一日毎にログファイルを切り替える
log_rotation_size = 10MB                            # 10MB を超過したら、ログファイルを新しく作成
```

postgresql.conf の設定を反映させます。今回、反映に再起動が必要なパラメータを修正したので、一度 PostgreSQL サーバを再起動します。

```
# su - postgres                                     (PostgreSQL 管理者ユーザでログインし直す)
$ pg_ctl restart                                    (PostgreSQL サーバを再起動する)
```

次に、logwatch をインストールします。もし logwatch がインストールされている場合、この処理は飛ばして構いません。

```
$ su -                                              (管理者でログイン)
# yum -y install logwatch                           (logwatch をインストール)
```

次に、pgsql_logwatch を適用します。pgsql_logwatch は GitHub 上で公開されていますので、ZIP アーカイブを入手し適用していきます。

```
# wget https://github.com/dalibo/pgsql_logwatch/archive/master.zip      (ZIP アーカイブを入手)
# unzip master                                                          (ZIP を展開)
```

pgsql_logwatch の各種ファイルを logwatch の適切なディレクトリに配置していきます。これらの手順については pgsql_logwatch の README に記載されています。

```
# cd pgsql_logwatch-master
# cp logfiles_postgresql.conf /etc/logwatch/conf/logfiles/postgresql.conf
# cp services_postgresql.conf /etc/logwatch/conf/services/postgresql.conf
# cp scripts_postgresql /etc/logwatch/scripts/services/postgresql
# chmod 755 /etc/logwatch/scripts/services/postgresql
```

引き続き、アタッチメント用の設定を行います。

/etc/logwatch/conf/logfiles/postgresql.conf

```
#########################################################
# Logfile definition for PostgreSQL
# File is to be placed in
#       /etc/logwatch/conf/logfiles/postgresql.conf
#########################################################

# What actual file?  Defaults to LogPath if not absolute path....
LogFile = /usr/local/pgsql/data/pg_log/*.log                    (PostgreSQL のサーバログを指定)

# Expand the repeats (actually just removes them now)
*ExpandRepeats
```

最後に、午前 4 時に logwatch が実行され、管理者のメールアドレスである「xxx@yyy.com」に送付するように、crontab -e を利用して、cron の設定を変更します。
ところで、logwatch がインストールされると、/etc/cron.daily/に、「0logwatch」という実行ファイルが追加されます。

したがって、/etc/cron.daily/配下の実行可能ファイルを午前４時に実行するようにするように、crontab を変更すればよいわけです。

```
# crontab -e
```

vi が起動するので、以下の追記部分を更新します。

```
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# (以下が追記部分)
MAILTO=xxx@yyy.com

# *  *  *  *  * command to be executed
  0  4  *  *  * run-parts /etc/cron.daily
```

以上の設定を行うことで、午前４時に前日のサーバログを管理者に送付することができます。

4.25. 不正アクセスの SNMP 通知

対応する PCI DSS 要件	10.6

不正アクセスに対して、SNMP 通知を行うことは、PostgreSQL の標準機能で実現することはできません。そのため、別の枠組みを利用する必要があります。

4.25.1. 実現手段

PostgreSQL の標準機能に加えて、ログを監視／収集する外部ツール、SNMP トラップの送付を行う外部ツールを組み合わせることで、本要件を満たすことができます。以下のような構成をとり、各ツールに適切なものを利用してください。

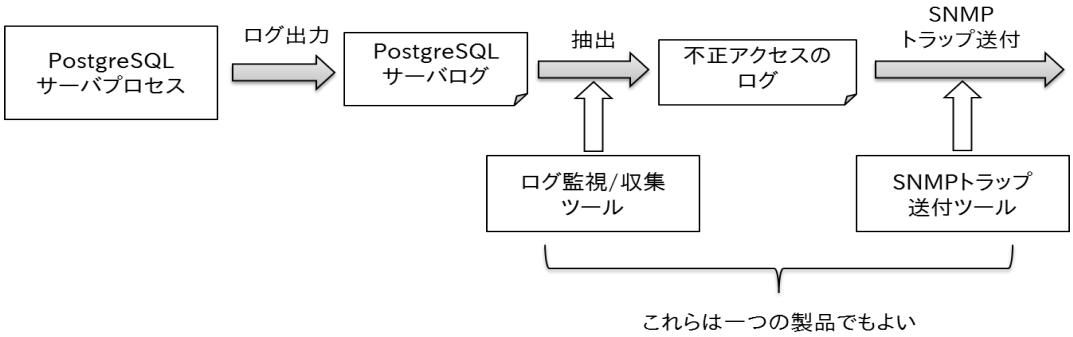


図 4.5: SNMP 通知の仕組み

4.25.2. ツールの例

ここでは、SNMP 通知が実現できる外部ツールについて紹介します。外部ツールの導入手順・利用方法については、各ツールのドキュメントや関連サイトをお読みください。

- rsyslog + omsnmp[28]
- fluentd[29] + 各種 fluentd 用プラグイン
- Zabbix[30]
- Hinemos[31]

  など

28 http://www.rsyslog.com/doc/omsnmp.html
29 http://www.fluentd.org/
30 http://www.zabbix.jp/
31 http://www.hinemos.info/

4.26. 不正アクセスを動的遮断する

対応する PCI DSS 要件	10.6

PostgreSQL では pg_terminate_backend 関数を利用することによって、手動で接続を動的に遮断することができますが、自動で切断する手段は標準機能にはありません。したがって、自動で不正アクセスを検知し、動的遮断を行う場合は、個別に工夫したスクリプトを作り定期実行して対処する必要があります。

4.26.1. 特定ユーザの接続を動的遮断する例

ここでは参考までに、遮断すべきユーザが分かっている場合に、動的遮断を行う方法について紹介します。

SELECT 文で pg_terminate_backend 関数を利用することで、バックエンドプロセスを動的に遮断することができます。

```
=# SELECT pg_terminate_backend( プロセス ID );
```

また、pg_stat_activity には、アクティブな接続に関する情報が格納されています。たとえば、以下のようなクエリで、確立している接続のユーザ名、バックエンドプロセス ID、IP アドレスを取得できます。

```
=# SELECT usename, pid, client_addr FROM pg_stat_activity;
 usename  | pid  |  client_addr
----------+------+---------------
 postgres | 1234 | 192.168.150.1
 user1    | 2345 | 192.168.150.2
 user2    | 2346 | 192.168.150.3
```

以上の pg_terminate_backend 関数と pg_stat_activity を組み合わせることで、特定のユーザや特定の接続を動的遮断することが可能です。例えば、user1 が不正アクセスだとわかった場合、以下のクエリで動的遮断が行えます。

```
=# SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE usename='user1';
```

4.27. ユーザアカウントごとのアクセス時間の定義

対応する PCI DSS 要件	10.6

アカウントごとにアクセスし得る正規の時間帯や曜日を定義することは、PostgreSQL の標準機能で行うことができません。したがって、代替手段を考える必要があります。
ここではアクセス時間の定義を、クライアント認証に用いる pg_hba.conf を複数用意し、cron で指定した時間や曜日で切り替えることにより実現する方法について紹介します。また、サーバログの接続情報を調べることで疑わしいアクセスを検知する方法を紹介します。

4.27.1. 実現手段

クライアント認証に用いる pg_hba.conf には、ユーザ名や接続元ホスト名によって、接続の許可・不許可を定義することができます。したがって、曜日・時間帯によって pg_hba.conf の内容を書き換えることができれば、ユーザごとにアクセス可能な時間を切り替えることが可能です。
これを実現する手順を以下に示します。

1. ユーザアカウントごとに、許可したいアクセス時間を定義する。
2. 1.をもとに、アクセス可能なユーザが切り替わるタイミングを割り出す。
3. 2.で割り出した、タイミングの 2 点の間を 1 つの接続ポリシーと定義する。
4. 接続ポリシーごとに pg_hba.conf の元となるファイルを作成する。なお、内容が重複するものは 1 つにしてよい。
5. pg_hba.conf を書き換える以下のスクリプトを作成する。
   - 4.で作成したファイルの中から、引数に指定した接続ポリシーを、pg_hba.conf に上書きする。
   - pg_ctl reload を実行して、pg_hba.conf を反映する。
6. crontab を更新し、1.で割り出したタイミングで、5 のスクリプトを実行しポリシーを切り替えるようにする。

4.27.2. 適用例

例として、ユーザアカウントのアクセス時間帯を以下のように定義するとします。

表 4.34: アクセス時間帯定義

ユーザ名	曜日・時間帯
user1	月曜日～金曜日：08:00～20:00、土曜日：08:00～18:00
user2	月曜日～金曜日：08:00～20:00、日曜日：08:00～18:00
postgres(管理者)	常時アクセス可能

ここでアクセス可能なユーザと対応する時間で区切ると、以下のように 4 パターンの接続ポリシーを設けることで実現できることが分かります。

表 4.35: 接続ポリシー一覧

ポリシー番号	アクセス可能なユーザ	対象となる時間帯
1	user1, user2, postgres	月曜日～金曜日：08:00～20:00
2	user1, postgres	土曜日：08;00～18:00
3	user2, postgres	日曜日：08;00～18:00
4	postgres	月曜日～金曜日： 00:00～08:00、20:00～24:00 土曜日：00:00～08:00、18:00～24:00 日曜日：00:00～08:00、18:00～24:00

したがって、クライアント認証に用いる pg_hba.conf を 4 種類用意し、時間帯によって切り替えればよいわけです。
まず、pg_hba.conf の元となるファイル pg_hba.conf.policyX(X はポリシー番号)を以下のように作成します。

$PGDATA/pg_hba.conf.policy1

```
# TYPE   DATABASE  USER     ADDRESS    METHOD
  host   all       user1    0.0.0.0/0  md5
  host   all       user2    0.0.0.0/0  md5
  host   all       postgres 0.0.0.0/0  md5
```

$PGDATA/pg_hba.conf.policy2

```
# TYPE   DATABASE  USER     ADDRESS    METHOD
  host   all       user1    0.0.0.0/0  md5
  host   all       postgres 0.0.0.0/0  md5
```

$PGDATA/pg_hba.conf.policy3

```
# TYPE   DATABASE  USER     ADDRESS    METHOD
  host   all       user2    0.0.0.0/0  md5
  host   all       postgres 0.0.0.0/0  md5
```

$PGDATA/pg_hba.conf.policy4

```
# TYPE   DATABASE  USER     ADDRESS    METHOD
  host   all       postgres 0.0.0.0/0  md5
```

次に、接続ポリシーを切り替えるためのスクリプトを作成します。

$PGDATA/rotate_policy.sh

```
#!/bin/sh

# 引数で指定した接続ポリシーを pg_hba.conf に上書き
cp -f $PGDATA/pg_hba.conf.$1 $PDGATA/pg_hba.conf

# 設定ファイルを読み直し、 pg_hba.conf を反映する。
pg_ctl reload
```

上記の接続ポリシー切り替え用スクリプトを、時間帯によって切り替えるように、crontab を編集します。

```
# su - postgres                                          (PostgreSQL 管理者ユーザでログインし直す)
$ crontab -e                                             (ユーザごとの crontab を編集する)
```

vi が起動するので、以下の追記部分を更新します。

```
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# *  *  *  *  * command to be executed
# (以下が追記部分)
  0  8  *  * 1-5 $PGDATA/rotate_policy.sh policy1
  0 20  *  * 1-5 $PGDATA/rotate_policy.sh policy4
  0  8  *  * 6   $PGDATA/rotate_policy.sh policy2
  0 18  *  * 6   $PGDATA/rotate_policy.sh policy4
  0  8  *  * 7   $PGDATA/rotate_policy.sh policy3
  0 18  *  * 7   $PGDATA/rotate_policy.sh policy4
```

以上の設定を行うことで、ユーザアカウントごとにアクセスできる時間を切り替えることが可能です。

4.28. アクセス時間外の接続検知

対応する PCI DSS 要件	10.6

4.27.「ユーザアカウントごとのアクセス時間の定義」でアクセス時間を定義した場合、アクセス時間外では、クライアント認証に用いる pg_hba.conf によって接続が拒否されます。したがって、サーバログの接続拒否の情報を検出することで、アクセス時間外の接続を検知することが可能です。

4.28.1. 実現手段

アクセス時間外の接続を検知できるようにするためには、以下の手順が必要です。

1. 4.27.「ユーザアカウントごとのアクセス時間の定義」の方法でアクセス可能時間を定義する。
2. サーバログに接続情報を出力するために、postgresql.conf の log_connections パラメータを on にする。
3. log_line_prefix を変更し、接続元のユーザ名や接続情報を含んだサーバログを出力するようにする。
4. ログを監視する枠組みを利用して、サーバログからアクセス時間外の接続を検知する。

4.28.2. 適用例

ここでは、サーバログをファイルに書き出し、「no pg_hba.conf entry」という文字列が含まれる行を grep によって抽出することで、アクセス時間外の接続を検知する方法を紹介します。

まず、postgresql.conf を以下のように指定します。

$PGDATA/postgresql.conf

```
# (中略)
# 接続情報をサーバログに出力、 サーバログの接頭を設定
log_connections = on                        # 接続をログに出力する(反映には再起動が必要)
log_line_prefix = '[%t][%p][%u][%d]'        # [日付時刻][プロセス ID][ユーザ名][データベース名]

# ログをファイルに書き出すように設定
log_destination = 'stderr'                      # 標準エラー出力に表示(反映には再起動が必要)
logging_collerctor = on                         # 標準エラー出力をログファイルに書き出す
log_directory = 'pg_log'                        # ログを書き出すディレクトリを指定
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log' # ファイル名のフォーマットを指定
log_rotation_age = 1d                           # 一日毎にログファイルを切り替える
log_rotation_size = 10MB                        # 10MB を超過したらログファイルを新しく作成
```

postgresql.conf の設定を反映させます。今回、反映に再起動が必要なパラメータを修正したので、一度 PostgreSQL サーバを再起動します。

```
# su - postgres                                     (PostgreSQL 管理者ユーザでログインし直す)
$ pg_ctl restart                                    (PostgreSQL サーバを再起動する)
```

pg_hba.conf によって接続に失敗したログを grep するようなスクリプトを作成します。ここでは前日の時間外アクセスを対象とします。

/home/postgres/badconnections.sh

```
#!/bin/sh

# 昨日の日付を YYYY-mm-dd 形式で取得
YESTERDAY = `date +" %Y-%m-%d"  -d '1 days ago'`

# $PG_DATA/pg_log 配下のログを検索し、 pg_hba に関連する FATAL エラーを出力する
cat $PGDATA/pg_log/postgresql-$(YESTERDAY)*.log | grep -E ".*FATAL.*pg_hba.*"
```

後は調べたいときに、以下のようにスクリプトを実行することで、クライアント認証によって接続拒否されたユーザアカウントを検知することができます。

```
$ cd ~
$ ./badconnections.sh
[2015-01-01 01:23:45 JST][12345][user1][database1]FATAL:  no pg_hba.conf entry for host
”[192.168.0.2]”, user “user1”, database “database1”
```

ただし注意点として、この方法では接続拒否されたユーザアカウントが、元々接続を許可されていないユーザか、時間外アクセスのために接続拒否されたユーザかは判断できません。本スクリプトで出力されたユーザ名と接続ポリシーで定義されているユーザ名との照合が別途必要です。

今回の user1 が時間外接続かどうかは、例えば接続ポリシーごとに分けた pg_hba.conf に対して grep コマンドなどを利用することで確認できます。user1 は接続ポリシー１と２で定義されているので、grep を実行したときに該当行が抽出されます。

```
$ grep ”user1” $PGDATA/pg_hba.conf.*
pg_hba.conf.policy1:  host   all       user1    0.0.0.0/0  md5
pg_hba.conf.policy2:  host   all       user1    0.0.0.0/0  md5
```

4.29. 格納データの暗号化

対応する PCI DSS 要件	6.3.1.3

格納するデータの暗号化は、PostgreSQL の contrib モジュールである pgcrypto で実現できます。

本節の暗号化を利用することにより、特定のテーブル/カラムを暗号化できます。ただし、アプリケーションからデータベース間や、データベースのレプリケーション間の NW で復号に必要なデータが流れる可能性があります。しかし、アプリケーションで暗号化するよりも PostgreSQL に暗号化ロジックや鍵管理を任せることができるので、開発への影響を少なくできます。

pgcrypto は共有鍵方式と、公開鍵方式を用意していますが、暗号化強度の観点から公開鍵方式がよいと考えます。また、暗号化/復号にはサーバの CPU リソースがかかりますので、必要最小限のカラムを暗号化してください。

以下に、pgcrypto の設定方法と暗号化/復号する例を紹介します。

4.29.1. PostgreSQL の設定

pgcrypto は contrib モジュールなので、rpm パッケージを利用して contrib モジュールを対象ホストにインストールします。

```
# sudo rpm -ivh postgresql93-contrib-9.3.5-1PGDG.rhel6.x86_64.rpm
warning: postgresql93-contrib-9.3.5-1PGDG.rhel6.x86_64.rpm: Header V4 DSA/SHA1 Signature,
key ID 442df0f8: NOKEY
Preparing...                ########################################### [100%]
   1:postgresql93-contrib   ########################################### [100%]
# rpm -qa | grep postgres
postgresql93-libs-9.3.5-1PGDG.rhel6.x86_64
postgresql93-contrib-9.3.5-1PGDG.rhel6.x86_64
postgresql93-9.3.5-1PGDG.rhel6.x86_64
```

PostgreSQL の対象データベースの pgcrypto を有効化します。

```
# psql database1
psql (9.3.5)
Type "help" for help.

database1=# CREATE  EXTENSION pgcrypto;
```

暗号化/復号するためのキーペアを生成します。
今回は GnuPG を用いて作成します。

```
# gpg --gen-key
gpg (GnuPG) 2.0.14; Copyright (C) 2009 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection?                                                    Enter
キーを押下
RSA keys may be between 1024 and 4096 bits long.
```

```
What keysize do you want? (2048)                                    Enter
キーを押下
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
       <n>  = key expires in n days
       <n>w = key expires in n weeks
       <n>m = key expires in n months
       <n>y = key expires in n years
Key is valid for? (0)                                               Enter
キーを押下
Key does not expire at all
Is this correct? (y/N) y                                            y 入力後、Enter
キーを押下

GnuPG needs to construct a user ID to identify your key.

Real name: postgres  # User 名を入力後、Enter キーを押下
Email address: postgres@example.com  # メールアドレスを入力後、Enter キーを押下
Comment:  # Enter キーを押下
You selected this USER-ID:
    "postgres <postgres@example.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O        O を入力後、Enter キーを押下
You need a Passphrase to protect your secret key.

gpg-agent[1922]: directory `/home/postgres/.gnupg/private-keys-v1.d' created
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
pub   2048R/8D12CAEA 2015-02-06
Key fingerprint = 34AC 2F93 A8F7 7E77 4CF2  360D D04E E841 8D12 CAEA
uid                  postgres <postgres@example.com>
sub   2048R/41EDF22A 2015-02-06
```

利用する鍵を管理するテーブルを作成し、生成した鍵を挿入します。この時、鍵文字列を入力するので PostgreSQL のログに SQL を出力する設定を OFF にしておくことが望ましいです。また、鍵を管理するテーブルへのアクセスは権限付与等により適切に行うことをご検討ください。

```
# psql database1
psql (9.3.5)
Type "help" for help.

database1=# CREATE TABLE key_table(public text, secret text, password text);
CREATE TABLE

database1=# ¥set PUBLIC `gpg -a --export`
database1=# ¥set SECRET `gpg -a --export-secret-keys`
database1=# INSERT INTO key_table VALUES(:'PUBLIC',:'SECRET','[pgp_key 生成時の
```

```
password]');
INSERT 0 1
```

4.29.2. 暗号化データの挿入

今回は簡単なユーザ情報のテーブルを想定して、情報を暗号化します。
以下のようなユーザ情報テーブルを作成します。暗号化対象のカラムは bytea 型にします。

```
# psql database1
psql (9.3.5)
Type "help" for help.

database1=# CREATE TABLE user_table(id int, user_name text, phone_num bytea);
CREATE TABLE
database1=# ¥d user_table
    Table "public.user_table"
  Column   |  Type   | Modifiers
-----------+---------+-----------
 id        | integer |
 user_name | text    |
 phone_num | bytea   |
```

対象のテーブルに暗号化したレコードを挿入します。対象カラムの値は pgp_pub_encrypto 関数を利用して、暗号化します。引数には挿入データ、公開鍵をとります。鍵は ASCII-Armor 形式になっているので dearmor 関数を利用して利用可能にします。

```
=# INSERT INTO user_table VALUES (1,'text user', pgp_pub_encrypt('08011112222',
dearmor((SELECT public FROM key_table))));
INSERT 0 1
```

挿入したデータは暗号化されたバイナリ形式の値になります。

```
=# SELECT * FROM user_table ;
-[ RECORD1 ]-----------------------------------------------------------------------------
id        | 1
user_name | text user
phone_num |
¥xc1c04c03e7d7d7c041edf22a0107ff626910e1e87fdf99a0f83982a218d5c902b17c1371325407231aeb1ab
b274b98464bde0233050c85c1cf72f2bb745e00e948c14e37b83aeb6cb151414ee4579f19f3c3869008af671b
0b551bdc22225f5f48b8fe9db5948b93917ddd153375f85cd372f907fd02a3da34dd5c5d85f1b1d7e6f92b4f0
e09b06dafa298e7065b90bcb942b6db9e9da5fa5152b06ad2d9206247459995dd9a4503979fd51b17dc5cc459
2cda13fe74fb2b7d8e164372c41ee8b987975952384ffcaf55735ce7af3eec8dc0375109b8a5ca90f0e234dbb
5fef8244ade4623919b07b99f9624d95075b2a383310f1350fcf6772d48622f87e6e24a130cd157849a368273
0729bc8476d23c01173d927a0b3096ae93be01d329ef9b718e95b9967ac9fbb42213e0a603c16be0bcdebde9f
591bc05c98b477b78b1f3632c35d592621a0283d59bf8
```

4.29.3. 暗号化データの復号

暗号化したデータを参照するには pgp_pub_decrypto 関数を利用して、復号します。引数には参照データ、秘密鍵、秘密鍵のパスワードをとります。この時、鍵管理テーブルから副問い合わせをすることで、PostgreSQL のログに鍵文字列が出力されないようにしています。

```
=# SELECT id,
         user_name,
         pgp_pub_decrypt(phone_num, dearmor((SELECT secret FROM key_table)),
                         (SELECT password FROM key_table))
```

```
   FROM   user_table ;
 id | user_name | pgp_pub_decrypt
----+-----------+-----------------
  1 | text user | 08011112222
(1 row)
```

このように、機密性の高いデータを暗号化することができます。

4.29.4. 運用上の注意点

データが暗号化されても、鍵の管理を正しくしないと情報漏えいにつながります。運用では以下のような点に気を付けてください。

- データバックアップは鍵管理テーブルは別にバックアップを取得して、管理場所を分ける。
  ex) pg_dump をテーブルごとに実施して、鍵管理テーブルだけは別ファイルにする。
- レプリケーション構成をとる場合は、盗聴の恐れのないとわかる安全な NW を利用する。

4.30. 格納データの透過的暗号化

対応する PCI DSS 要件	6.3.1.3

Transparent Data Encryption for PostgreSQL（以下、TDE)は PostgreSQL 専用の暗号化ツールであり、PostgreSQL データベースサーバに暗号化データ型を追加します。NEC が作成した拡張モジュールです。
業務アプリケーションを改修することなくデータの暗号化が可能であるため、比較的簡単にシステムのセキュリティが向上します。pgcrypto の暗号化機能をベースとして実装されています。

Free Edition(無償版)と Enterprise Edition(有償版)があります。

- Transparent Data Encryption for PostgreSQL Free Edition
  GPLv3 ライセンスで公開されています。TDE の基本機能が実装されています。

- Transparent Data Encryption for PostgreSQL Enterprise Edition
  Free Edition に加えて、対象データ型の追加、データベース診断機能、データベース復旧機能、暗号鍵管理機能等を搭載した製品版です。

以下は Free Edition について説明します。

4.30.1. 導入

以下はソースからコンパイルする手順となります。
rpm も提供されていますので、対象 OS や導入手順については GitHub 上に掲載されているマニュアルを参考にしてください。

1. tar ボールのダウンロードおよびサーバへのアップ、tar ボールの解凍
2. cofigure、コンパイル、リンク
3. パラメータ設定および再起動
4. pgcrypto エクステンションの作成
5. TDE セットアップ（データベース単位）
6. 暗号化アルゴリズムおよび鍵の設定（データベース単位）

実行例を記載します。PostgreSQL 9.4 および TDE 1.1.1.1 を使用しています。
なお執筆時点(2016/02)で対応している PostgreSQL のバージョンは 9.3 および 9.4 です。

1. tar ボールのダウンロードおよびサーバへのアップ、tar ボールの解凍
   Github で公開[32]されている tar ボールをダウンロードし、導入対象の PostgreSQL へアップロードします。

   （解凍例）

```
$ unzip tdeforpg-master.zip
Archive:  tdeforpg-master.zip
e1984e696ed219397489ebdbff060f293393490b
   creating: tdeforpg-master/
  inflating: tdeforpg-master/.gitignore
  inflating: tdeforpg-master/COPYRIGHT
  inflating: tdeforpg-master/INSTALL-NOTE.TXT
  inflating: tdeforpg-master/LICENSE
  inflating: tdeforpg-master/README
  inflating: tdeforpg-master/RELEASE-NOTE.TXT
   creating: tdeforpg-master/SOURCES/
...
```

32 https://github.com/nec-postgres/tdeforpg

```
finishing deferred symbolic links:
  tdeforpg-master/SOURCES/data_encryption/94 -> 93
$
```

2. コンパイル、リンク
   次の環境変数を設定します。

表 4.36: TDE 導入に必要な環境変数

環境変数	設定
TDEHOME	解凍した TDE のソースツリーのディレクトリ
PGSRC	PostgreSQL 導入時のソースツリーのディレクトリ
PGHOME	PostgreSQL のインストールディレクトリ

（実行例）コンパイル

```
$ export TDEHOME=~/media/tdeforpg-master
$ export PGSRC=~~/media/postgresql-9.4.1
$ export PGHOME=~/p941_home

$ cd $PGSRC
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking which template to use... linux
…
config.status: linking src/makefiles/Makefile.linux to src/Makefile.port
$ echo $?
0

■pgcrypto のライブラリを/usr/lib64 ディレクトリに対してシンボリックリンク作成

$ su
パスワード:
# ln -s $PGHOME/lib/pgcrypto.so /usr/lib64/libpgcrypto.so
# exit

■TDE のライブラリ作成

$ cd $TDEHOME/SOURCES/data_encryption
$ sh makedencryption.sh 94 $PGSRC
rm -f data_encryption.so data_encryption.o
/usr/bin/gcc -c -Wall -O3 -fPIC -I/home/p941/media/postgresql-9.4.1/src/include
-I/home/p941/media/postgresql-9.4.1/contrib/pgcrypto data_encryption.c
/usr/bin/gcc -shared -o data_encryption.so data_encryption.o
-L/home/p941/media/postgresql-9.4.1/contrib/pgcrypto -lpgcrypto -Wl,-
rpath,'/home/p941/media/postgresql-9.4.1/contrib/pgcrypto'
        linux-vdso.so.1 =>  (0x00007fff1d5ff000)
        libpgcrypto.so => /usr/lib64/libpgcrypto.so (0x00007fcda9ad6000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fcda9741000)
        libz.so.1 => /lib64/libz.so.1 (0x00007fcda952b000)
        /lib64/ld-linux-x86-64.so.2 (0x0000003fa4200000)

INFO: data_encryption.so was made.

$ cd $TDEHOME/SOURCES/data_encryption/94
$ ls -l
合計 84
-rw-r--r--. 1 p941 postgres   757 12月 10 11:22 2015 Makefile
-rw-r--r--. 1 p941 postgres 22999 12月 10 11:22 2015 data_encryption.c
-rw-r--r--. 1 p941 postgres  1255 12月 10 11:22 2015 data_encryption.h
-rw-r--r--. 1 p941 postgres 22128  2月 25 00:02 2016 data_encryption.o
-rwxr-xr-x. 1 p941 postgres 25291  2月 25 00:02 2016 data_encryption94.so.1.1.1.1

■TDE のライブラリを/usr/lib64 ディレクトリに対してシンボリックリンク作成

$ su
```

```
パスワード:
# ln -s $TDEHOME/SOURCES/data_encryption/94/data_encryption94.so.1.1.1.1
 /usr/lib64/data_encryption.so
```

3. パラメータ設定および再起動
   postgresql.confに以下を設定します。

```
shared_preload_libraries = '/usr/lib64/data_encryption.so'       --- TDE のライブラリ
```

PostgreSQL の再起動で反映し、確認します。

```
$ pg_ctl restart -m fast -w

$ psql

=# show shared_preload_libraries;
   shared_preload_libraries
------------------------------
 /usr/lib64/data_encryption.so
(1 行)
```

4. pgcrypto エクステンションの作成
   TDE をセットアップするデータベースに対して実行します。対象データベースが複数の場合は、それぞれに対して実行します。

```
=# CREATE EXTENSION pgcrypto;
CREATE EXTENSION

=# ¥dx
                                    インストール済みの拡張の一覧
        名前        | バージョン |  スキーマ  |           説明
--------------------+------------+------------+--------------------------------------
 pgcrypto           | 1.1        | public     | cryptographic functions
  ～略～
```

5. TDE セットアップ（データベース単位）

   セットアップシェルを実行します。以下の入力を求められます。
   データベース毎に行います。対象が複数ある場合は、それぞれに実行します。

表 4.37: セットアップシェルの入力項目

プロンプト	入力
Transparent data encryption feature setup menu 1: activate the transparent data encryption feature 2: inactivate the transparent data encryption feature	TDE を有効化するため 1 を指定
Please enter database server port to connect :	PostgreSQL のポート番号　: 5432
Please enter database user name to connect : Please enter password for authentication :	ユーザ名（スーパーユーザ）: postgres パスワード　: xxxxxxxx
Please enter database name to connect :	TDE をセットアップするデータベース : postgres

（実行例）

```
$ cd $TDEHOME/SOURCES
$ sh bin/cipher_setup.sh $PGHOME
Transparent data encryption feature setup script
Please select from the setup menu below
Transparent data encryption feature setup menu
1: activate  the transparent data encryption feature
2: inactivate the transparent data encryption feature
select menu [1 - 2] > 1
```

```
Please enter database server port to connect : 5432
Please enter database user name to connect : postgres
Please enter password for authentication :
Please enter database name to connect : postgres
CREATE LANGUAGE
INFO: Transparent data encryption feature has been activated
$
```

6. 暗号化アルゴリズムおよび鍵の設定(データベース毎)
   暗号化アルゴリズムおよび鍵の設定シェルを実行します。以下の入力を求められます。
   データベース毎に行います。対象が複数ある場合は、それぞれに実行します。

表 4.38: 暗号化アルゴリズムおよび鍵の設定シェルの入力項目

プロンプト	入力
Please enter database server port to connect :	PostgreSQL のポート番号　: 5432
Please enter database user name to connect : Please enter password for authentication :	ユーザ名(スーパーユーザ) : postgres パスワード　: xxxxxxxx
Please enter database name to connect :	TDE をセットアップ済みのデータベース : postgres
Please enter the new cipher key : Please retype the new cipher key :	暗号化鍵
Please enter the algorithm for new cipher key :	暗号化アルゴリズム : aes

(実行例)

```
$ sh bin/cipher_key_regist.sh $PGHOME
=== Database connection information ===
Please enter database server port to connect : 5432
Please enter database user name to connect : postgres
Please enter password for authentication :
Please enter database name to connect : postgres
=== Regist new cipher key ===
Please enter the new cipher key :
Please retype the new cipher key :
Please enter the algorithm for new cipher key : aes

Are you sure to register new cipher key(y/n) : y
$
```

暗号化アルゴリズムは次から選択します。通常は aes を指定します。

表 4.39: TDE で選択可能な暗号化アルゴリズム

暗号化指定	暗号化アルゴリズム	暗号化後データサイズ
aes	AES128	((<元データサイズ> / 16) + 1) * 16 + 2
bf	Blowfish	((<元データサイズ> / 8) + 1) * 8 + 2

public.cipher_key_table テーブルを参照して鍵とアルゴリズムが設定されている事を確認します。
key 列はソルト付きの AES256 にて暗号化されています。

```
=# ¥x auto
拡張表示が自動的に使用されます

=# SELECT * FROM public.cipher_key_table;
-[ RECORD 1 ]---------------------------------------------------------------------------
key       | ¥xc30c040901024fc68d7b69ab5ce1d23401162a124cf537c9ecb3a271192eb46bdd88e9971
          | 6281732fe89c40682ffaaf4087945ce2c10ed8d0b414c1497b0cc86c60ad47a
algorithm | aes
```

4.30.2. 暗号化用のデータ型

TDE ではセットアップシェルにより、以下の暗号化用のデータ型が実装されます。
char 型,varchar 型,text 型などの文字型は encrypt_text 型を使用します。
Free Edition では日付型や数値型は実装されません。

表 4.40:暗号化用のデータ型

データ型	格納データ	対応するデータ型
encrypt_text	テキストデータ用	char 型,varchar 型,text 型
encrypt_bytea	バイナリデータ用	bytea 型

（テーブル作成例）ename 列のデータ型に encrypt_text を使用

```
=# CREATE TABLE tde_emp (empno numeric,ename encrypt_text);
CREATE TABLE

=# ¥d tde_emp
   テーブル "public.tde_emp"
  列   |      型       | 修飾語
-------+--------------+--------
 empno | numeric      |
 ename | encrypt_text |
```

4.30.3. TDE セッション開始関数

TDE を使用するには、セッション毎に以下の関数を実行する必要があります。

表 4.41: TDE セッション開始時に必要な処理

関数	パラメータ	機能
cipher_key_disable_log	なし	pgtde_begin_session 関数実行時に、サーバログに暗号化鍵がマスキング化して出力されます。暗号化鍵の漏洩を防ぐのが目的です。 戻り値は boolean 型で結果の成否を戻します。
pgtde_begin_session	暗号化鍵	暗号化セッションを開始する。パラメータで暗号化鍵を指定します。 戻り値は boolean 型で結果の成否を戻します。 指定した暗号化鍵が cipher_key_regist.sh にて登録した文字列と異なる場合はエラーとなります。

（実行例）

```
■暗号化鍵のサーバログへの出力をマスキング

=# SELECT cipher_key_disable_log();
 cipher_key_disable_log
------------------------
 t

■暗号化セッション開始

=# SELECT pgtde_begin_session('tde');
 pgtde_begin_session
---------------------
 t
```

log_statement=all の場合、pgtde_begin_session 関数実行時にはサーバログ(csvlog)に以下が出力されます。パラメータとして入力した文字列がマスキングされている事が確認できます。

（サーバログ出力例）

```
LOG,00000,"statement: SELECT pgtde_begin_session(*****);",,,,,,,,,"psql"
```

4.30.4. TDE 実行例

TDE セッションを開始した後は、暗号化／復号化を意識せずに SQL を実行できます。

実行例）

```
■TDE データ登録

=# INSERT INTO tde_emp VALUES (1000,'鈴木一郎');
INSERT 0 1

■TDE データ参照

=# SELECT * FROM tde_emp ;
 empno |     ename
-------+---------------
  1000 | 鈴木一郎
```

TDE データに対して TDE セッション開始関数を実行しないで参照を試みると、以下の様なエラーが発生します。

```
■新規セッション

=# ¥connect postgres postgres
データベース "postgres" にユーザ"postgres"として接続しました。

■TDE データ参照

=# SELECT * FROM tde_emp ;
ERROR:  TDE-E0017 could not decrypt data, because key was not set[01]
```

サーバログ(csvlog)には以下の出力があります。

（サーバログ出力例）

```
ERROR,58030,"TDE-E0017 could not decrypt data, because key was not set[01]",,,,,,"SELECT *
FROM tde_emp ;",,,"psql"
```

PL/pgSQL で実行した場合は、SQLSTATE 58030,"IO_ERROR"例外が発生します。

```
EXCEPTION
    WHEN IO_ERROR THEN
        ...
```

4.30.5. TDE データのインデックス

暗号化データによるインデックス作成には以下の考慮が必要です。

- 暗号化データは元データとバイト順序が異なるため、BTREE インデックスによる範囲検索はできません。
- 復号化データによるインデックス作成は情報漏えいの恐れがあります。

このため TDE では HASH インデックスによる完全一致検索のみサポートされています。
暗号化データにより作成するため、インデックス作成時には TDE セッションを開始する必要はありません。

（インデックス作成例）

```
=# CREATE INDEX tde_emp_idx1 ON tde_emp USING HASH (ename);
CREATE INDEX
```

（インデックス使用例）

```
■TDE セッション開始

=# SELECT cipher_key_disable_log();
 cipher_key_disable_log
------------------------
 t

=# SELECT pgtde_begin_session('tde');
 pgtde_begin_session
---------------------
 t

■完全一致検索（索引使用）

=# EXPLAIN SELECT * FROM tde_emp WHERE ename = '鈴木一郎';
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tde_emp  (cost=36.13..99.20 rows=17 width=64)
   Recheck Cond: (ename = '鈴木一郎'::encrypt_text)
   ->  Bitmap Index Scan on tde_emp_idx1  (cost=0.00..36.13 rows=17 width=0)
         Index Cond: (ename = '鈴木一郎'::encrypt_text)

■前方一致検索（索引使用不可）

=# EXPLAIN SELECT * FROM tde_emp WHERE ename LIKE '鈴木%';
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tde_emp  (cost=0.00..1728.45 rows=17 width=64)
   Filter: ((ename)::text ~~ '鈴木%'::text)
```

4.30.6. 注意点

1. 暗号化アルゴリズムは AES の他に BlowFish もありますが、暗号化/復号化のパフォーマンスが AES に劣るため、あまりお奨めできません。

2. TDE は基本的にはアプリケーションから透過的に使用できますが、TDE セッション開始関数はアプリケーションに実装する必要があります。

3. 暗号化鍵は変更可能です。その際に既存の暗号化データは再暗号化されるため、データ量に応じた時間が掛かります。

4. 暗号化はパフォーマンスに対して一定の影響があります。
   - TDE では pgcrypto の暗号化機能をベースとしているため、同等の負荷を想定して下さい。
   - 暗号化列の選定にあたってはデータの重要性を考慮の上ご検討下さい。
   - 十分なパフォーマンス検証を行う事をお奨めします。

4.31. ファイルシステム透過的暗号化

対応する PCI DSS 要件	3.4.1、3..5、3.5.1、3.5.2

PostgreSQL 本体には透過的に暗号化してデータを格納する機能がありません。対処策の一つとして、OS レベル(ファイルシステム)の機能を使う方法があります。本項目では、Linux で標準的に利用されている dm-crypt 暗号化機能について紹介します。また、透過的暗号化機能の一般的な説明も記載します。

4.31.1. ファイルシステム透過的暗号化とは

ファイルシステム透過的暗号化とは、ファイルシステムのレイヤで自動的に暗号化して格納し、復号して読み出す仕組みです。データベースクライアントのアプリケーションには何ら変更を加えずに適用することができます。データベースクライアントアクセスを通じて、暗号化していない場合と同様にデータの読み書きができ、かつ適切な手順を踏まない限りデータを読むことができない、ということが実現できます。これにより、ディスクの抜き取りや仮想イメージデータの奪取などのデータ盗み出しに対するデータ保護が可能になります。

なお、データへのアクセスについては、通常のファイルシステムを利用するのと変わりませんが、マシンや OS の再起動時にパスフレーズを入力しなければいけない等、運用面で異なる点があることに注意してください。

4.31.2. ファイルシステム透過的暗号化の適用手順

dm-crypt 暗号化機能を用いたファイルシステム透過的暗号化機能の具体的な使い方手順についてまとめます。

1)暗号化の設定

下記コマンドにより、デバイスの暗号化を設定します。下記例では、luksFormat による暗号化を行うためのパスワード設定を行っています。

```
# cryptsetup luksFormat /dev/sdb2

WARNING!
========
This will overwrite data on /dev/sdb2 irrevocably.

Are you sure? (Type uppercase yes): YES
Enter LUKS passphrase:
Verify passphrase:
```

2)暗号化デバイスをオープン

OS が認識できるように、暗号化デバイスをオープンします。

下記の例では、pgecons という名称のデバイスとして認識されます。

```
# cryptsetup luksOpen /dev/sdb2 pgecons
Enter passphrase for /dev/sdb2:
```

以降、上記コマンドを実行しないで直接デバイスにマウントしようとすると、下記のエラーとなります。

```
# mount /dev/sdb2 /secpgdata
mount: unknown filesystem type 'crypto_LUKS'
```

問題なくオープンできていると、/dev/mapper 配下に pgecons が作成されます。

```
# ls /dev/mapper/pgecons
```

/dev/mapper/pgecons

3)ファイルシステムの作成
認識したデバイス(/dev/mapper/pgecons)にファイルシステムを構築します。

```
# mkfs.ext4 /dev/mapper/pgecons
mke2fs 1.41.12 (17-May-2010)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
130048 inodes, 520064 blocks
26003 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=67633152
64 block groups
8192 blocks per group, 8192 fragments per group
2032 inodes per group
Superblock backups stored on blocks:
   8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409

Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 23 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

4)ファイルシステムへのマウント

```
# mount /dev/mapper/pgecons /secpgdata
```

以降、通常のデバイス同様に利用できます。下記では/secpgdata/data にデータベースクラスタを作成し、PostgreSQL を起動しています。

```
# su - postgres

$ initdb -D /secpgdata/data --no-locale -E UTF8
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "C".
The default text search configuration will be set to "english".

Data page checksums are disabled.

creating directory /secpgdata/data ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
```

```
$ pg_ctl -D /secpgdata/data start
server starting

$ createdb secdb

$ psql secdb -c "create table tbl(t text)";
CREATE TABLE

$ psql secdb -c "insert into tbl values ('test')";
INSERT 0 1
```

(5)ファイルシステムのアンマウント
アンマウント手順は通常のデバイスと同様に実施できます。

```
# umount /secpgdata/
```

(6)暗号化デバイスのクローズ
下記コマンドで暗号化デバイスをクローズできます。

```
# cryptsetup luksClose pgecons
```

4.32. PostgreSQL を拡張した商用製品による透過的暗号化

対応する PCI DSS 要件	3.4.1、3..5、3.5.1、3.5.2

PostgreSQL 本体には透過的暗号化をしてデータ格納する機能がありません。対処策の一つとして、PostgreSQL を拡張して透過的暗号化機能を付加した商用互換製品を使う方法があります。本項目では、透過的暗号化機能を備えた PowerGres Plus 製品について紹介します。また、透過的暗号化機能の一般的な説明も記載します。

4.32.1. 透過的暗号化とは

透過的暗号化とは、データベースサーバ側で自動的に暗号化して格納し、復号して読み出す仕組みです。データベースクライアントのアプリケーションには何ら変更を加えずに適用することができます。データベースクライアントアクセスを通じて、暗号化していない場合と同様にデータの読み書きができる一方で、データベースサーバ上のデータファイルを盗み出しても、そのままでは読むことができない、ということが実現できます。

暗号化、復号には鍵データを使用し、データベースサーバを起動したあと、データベース上の暗号化された領域にアクセス可能とするためには、鍵データを与える必要があります。データベースソフトウェアで透過的暗号化機能を実現するには、鍵データをどこに保管して、どのように利用するかということが課題となります。鍵データ自体が通常のストレージに平文で置かれていては意味が無いからです。通常、以下の方法がとられます。

1. 鍵データ自体を暗号化しておき、起動時または暗号化データの利用を有効化する時には人手でパスフレーズを入力して（あるいは所定の USB 機器を挿入して）、鍵データを復号して利用する。

2. セキュリティチップあるいは HSM（ハードウェアセキュリティモジュール）を使用して鍵データをそれらの領域に保管する。あるいは、それらチップ・機器の機能により鍵データを暗号化・復号して利用する。

3. 鍵データを何らかサーバマシン固有の情報や正規の稼働場所でないと得られない固有の情報、と結びつけた形で暗号化しておき、復号して利用する。

上記２番目と３番目は、いずれも、人手によるパスフレーズ入力を無くす一方で、物理的なデーター式を盗み取られた場合にデータを復号できないようにするための対策となります。上記２番目のセキュリティチップは TPM（Trusted Platform Module）として共通規格化され、主要なメーカーのサーバ製品に搭載あるいはオプションとして搭載可能となっています。また、ハードウェアセキュリティモジュール（HSM）とは、主にアプライアンス製品の形で提供される鍵管理・暗号処理専用のハードウェアです。上記３番目は、２番目が使えない場合の代替策といえます。

4.32.2. PowerGres Plus における透過的暗号化

PowerGres Plus 製品の総合的な案内・解説は製品 Web サイト[33]やマニュアル[34]を参照いただくものとし、ここでは透過的暗号化機能について、採用されている方式と、できること・できないことを表にまとめて記載します。
なお、PowerGres Plus で透過的暗号化機能が使用できるのは V9.1 バージョン以降となります。

表 4.42: PowerGres の透過的暗号化機能

暗号化単位	テーブルスペースごと
暗号化対象	以下要素が暗号化される。 (1) 暗号化対象のテーブルスペースにおけるテーブル・インデックス等のデータ格納ファイル (2) 暗号化対象のテーブルスペース内のオブジェクトの更新に対応した WAL データ (3) オンラインバックアップで取得するベースバックアップと WAL アーカイブ
暗号方式	２層の暗号化で構成される。以下の仕様となっている。

33 http://powergres.sraoss.co.jp/s/ja/product/Plus.php
34 http://powergres.sraoss.co.jp/manual/Plus/V91/linux/index.html

	・テーブルスペースは AES128、AES256 にて暗号化 ・テーブルスペース暗号化の鍵データは AES256 でマスタ暗号化キーで暗号化されファイル保存 ・マスタ暗号化キーは PBKDF2 にて暗号化されてファイル保存され、復号にパスフレーズ入力を必要とする
TPM・HSM 対応	無し。
AES-NI 対応	有り。 Intel Xeon プロセッサの 5600 番台以降に搭載された AES-NI 機能に対応しています。CPU の AES-NI 機能を使った場合、データ投入におけるオーバーヘッドは 10%以下程度、OLTP におけるオーバーヘッドは 3%以下程度、という性能検証結果が公表されている。
レプリケーション対応	ストリーミングレプリケーションを構成可能です。プライマリ、スタンバイとも、どちらも同様に透過的暗号化が適用されている状態となる。
リリースバージョン	PostgreSQL 9.4.x をベースにした PowerGres Plus V9.4 がリリースされてます(2016/04/01 現在)。

PowerGres Plus は透過的暗号化機能を、PostgreSQL の機能を制限することなく、また、小さい性能オーバーヘッドで実現できているといえます。欠点としては、対応 PostgreSQL バージョンが限定されることと、鍵データに TPM/HSM を利用できないことを挙げることができます。

4.32.3. 高可用クラスタ対応とマスター暗号化キー自動オープン

PowerGres Plus で共有ディスクやミラーリングブロックデバイスを使った高可用性クラスタを構成する場合には、どうやって起動時にパスフレーズを入力するか、という課題があります。自動の障害切り替えを実現するには、高可用性クラスタリングソフトウェアが PowerGres Plus を自動的に起動できなければいけません。

PowerGres Plus は「マスター暗号化キーの自動オープン」という機能を持っており、これを使用することで解決できます。マスタ暗号化キーの自動オープンを設定する場合、パスフレーズが、サーバマシン固有の情報をキーとして仕様非公開の固有方式にて暗号化されてファイル保存されて、これが起動時に自動で使用されます。

4.32.4. PowerGres Plus による透過的暗号化の適用手順

PowerGres Plus による透過的暗号化機能の具体的な使い方手順については、PowerGres 製品 Web サイトのチュートリアル記事「PowerGres 体験記 第 3 回 データを暗号化してみよう」の [PowerGres Plus の透過的データ暗号化] の項[35] が参考になります。

35 http://powergres.sraoss.co.jp/s/ja/tech/exp/plusv91/03_tde.php#tde

4.33. 長時間アイドル中の接続を自動切断する

対応する PCI DSS 要件	8.5.15

PostgreSQL 本体には、接続（セッション）がアイドル状態のまま一定時間経過したときに、自動的に切断する機能はありません。本項では、この機能を実現する代替策を示します。

4.33.1. 定期実行スクリプトによる方法

以下のようなスクリプトを cron に登録して、1 分毎～5 分毎くらいにて、繰り返し自動実行することで、接続して 15 分以上アイドル状態であるものを切断できます。

（スクリプト例、/usr/local/bin/db15min_idle_check.sh であるものとします）

```
#!/bin/sh
export PATH=/usr/pgsql-9.3/bin:$PATH
export LD_LIBRARY_PATH=/usr/pgsql-9.3/lib:$LD_LIBRARY_PATH
LOGFILE=/var/lib/pgsql/db15min_idle_check.log

cat <<'EOS' | psql -U postgres -d postgres -q -t >> $LOGFILE 2>&1

SELECT pg_terminate_backend(pid), pid, datname, usename, application_name,
  client_addr, client_hostname, client_port, query,
  backend_start, xact_start, query_start, state_change
  FROM pg_stat_activity
  WHERE pid <> pg_backend_pid() AND state LIKE 'idle%'
    AND state_change < current_timestamp - '15 min'::interval;

EOS
```

（crontab 設定例、5 分おきの場合）

```
*/5 * * * *     /usr/local/bin/db15min_idle_check.sh 2>&1 >/dev/null
```

（出力例、下記は実際には 2 行の出力）

```
 t                    | 4973 | database1      | user1    | psql             |
 192.168.150.5 |                 | 39104       | SELECT * FROM table1; |
2015-01-23 17:40:31.431766+09 | 2015-01-23 17:42:54.253215+09 | 2015-01-23 17:42:54.253215+09
| 2015-01-23 17:42:56.365212+09
 t                    | 4983 | database2      | user1    | psql             |
 192.168.150.5 |                 | 39105       | SELECT * FROM table2; |
2015-01-23 17:30:22.225178+09 | 2015-01-23 17:42:31.823251+09 | 2015-01-23 17:42:50.061413+09
| 2015-01-23 17:42:51.321214+09
```

出力されるカラムの意味は以下の通りです。

1. pg_terminate_backend 関数の結果 (成功すれば t)
2. 停止した接続のバックエンドプロセスの PID
3. データベース名
4. ユーザ名
5. 切断した接続のアプリケーション名
6. 切断した接続のアクセス元アドレス
7. 切断した接続のアクセス元ホスト名
8. 切断した接続のアクセス元ポート番号
9. 切断した接続で最後に実行していた SQL
10. 切断した接続の開始時刻
11. 切断した接続での最後のトランザクション開始時刻
12. 切断した接続での最後の SQL 実行開始時刻
13. 切断した接続で状態が idle または idle in transaction になった時刻

本スクリプトは、データベーススーパーユーザで接続して、pg_stat_activity ビューで他の接続の状態を調べて、アイドル状態が長ければ pg_terminate_backend() 関数を使って切断します。また、実際に切断を行った場合には、その接続の情報を出力します。これら情報は不正とみられるアクセスの調査に利用できます。

本スクリプトはローカル接続においてインタラクティブなパスワード入力無しで、postgres ユーザが接続できることを前提としてます。pg_hba.conf ファイルで trust と設定するか、パスワードファイル（~/.pgpass ファイル）を記述するか、いずれかの対応が必要です。

適用が簡単であるという点で優れた方法といえます。実際には、システムの実情に合わせて、スクリプト中の SELECT コマンド部分に例外とする条件を追加して、適用することが想定されます。

4.33.2. 接続プロキシソフトウェアによる方法

PostgreSQL 接続用のプロキシソフトウェアが存在します。良く知られているのは以下の 2 つです。

- pgpool-II
- pgbouncer

これらは PostgreSQL サーバと PostgreSQL クライアントの間を接続を仲介して動作して、コネクションキャッシュ等の機能を提供します。そこで提供される機能の一つとしてアイドル状態の接続を指定したタイムアウト時間にて自動で切断するというものがあります。pgpool-II、pgbouncer とも、この機能を備えています。

以下では pgpool-II を使った設定例を示します。

PostgreSQL が動作しているサーバ上に pgpool-II をインストールします。本項記載のための動作確認には以下の環境を使用しました。ただし、本機能は PostgreSQL バージョンを選びません。また、使用する pgpool-II の機能は、かなり古いバージョンから備わっていた機能であり、こちらも事実上バージョンを選ばないものといえます。

（本項記載における動作確認の環境）

```
OS: CentOS 6.4 x86_64
PostgreSQL 9.3.5
pgpool-II 3.3.4（ソースコードから）
```

pgpool-II のインストールにあたり、設定ファイル pgpool.conf に以下の設定を与えます。以下の囲み内には、付属する pgpool.conf.sample から変更すべき点と注意すべき点のみを記載しています。

（pgpool.conf の pgpool,conf.sample からの変更箇所および注意すべき点）

```
listen_addresses = '*'
port = 5433
backend_hostname0 = 'localhost'
backend_port0 = 5432
backend_weight0 = 1
num_init_children = 32
client_idle_limit = 300
pid_file_name = '/var/run/pgpool/pgpool.pid'
connection_cache = on
```

「port = 5433」としていますので待ち受けポートは 5433 になります。クライアントが pgpool-II の 5433 ポートに接続すると、同ホストで 5432 ポートで待ち受けしている PostgreSQL に接続されます。アプリケーションの接続先を、5433 ポートに変えることになります。「connection_cache = on」設定にてコネクションプーリングがデフォルトで有効になっていますので、不要でしたら、ここは off にしてください。また、「num_init_children」が同時接続数です。必要応じて調整してください。

「client_idle_limit = 300」がアイドル状態が 300 秒(5 分) 以上続いたら切断するという設定になります。アイドル状態が設定時間を超えた場合、以下のような pgpool-II のログメッセージが出力されます。PostgreSQL サーバ側には正常な切断動作が行われますので、PostgreSQL 側に特別なログメッセージは生じません。

（client_idle_limit 設定が動作したときの popool-II のログ）

```
2015-01-25 16:25:34 LOG:   pid 11234: pool_process_query: child connection forced to terminate
due to client_idle_limit (300) reached
```

クライアント側には、サーバ側からの突然の切断として見えます。psql であれば以下のようなエラーが生じることになります。末尾で「Attempting reset: Succeeded.」と出ているのは、psql は、サーバ側から切断されたとき、同ユーザ、同パスワードで自動で再接続を試みるようになっているためです。

(切断された psql 側のエラー例)

```
$ psql -p 5433 -h dbhost1 -d database1 -U user1
=>

 ・・・ 5分 アイドル状態を続けた後  ・・・

=> SELECT * FROM table2;
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
```

アイドル状態の自動切断に pgpool-II（ほか接続プロキシソフトウェア）を使うメリットと注意点は以下になります。

- 要求事項通りに機能が実現できる。
- コネクションプーリングやクエリキャッシュなどその他の機能の恩恵も受けることができる。
- 別ソフトウェアを追加で導入することになるため、習得すべき技術と管理コストが増える。
  - 接続経路が変るため、アクセス制限の設定の仕方も変る。例えば、pg_hba.conf で接続元ホストに制限を加えていても、すべてプロキシソフトウェアからの接続になってしまうので、同様の設定を今度はプロキシソフトウェア側、あるいは、ファイアウォールに記述することになる。
  - 同時接続数などもプロキシソフトウェア側の設定も必要となる。
  - 可用性という観点ではプロキシソフトウェアと PostgreSQL のどちらかがダウンするとデータベースのサービスダウンとなってしまう。

4.33.3. L4 ロードバランサによる方法について

L4 ロードバランサ装置（あるいはソフトウェア）は、前節で紹介したプロキシソフトウェア同様に PostgreSQL サーバと PostgreSQL クライアントの間を接続を仲介します。また、多くの装置で通信が無い状態が一定時間継続した場合に、自動でコネクションを切断する機能を有しています。

しかしながら、L4 ロードバランサを使う方法は適切ではありません。なぜなら、通信が無い状態と、PostgreSQL 接続におけるアイドル状態は異なるからです。応答に長時間かかる SQL の応答を待っている状態であっても、区別なく中途で切断されてしまうといった不具合をひき起こしてしまいます。

5. 検証結果

5.1. 目的

4 章までの調査・検証の観点は、セキュリティ機能について実施可能か否かでありました。
しかし、実際のシステムで稼働させるには、オーバーヘッドや運用への影響が懸念されます。
そこで性能や運用性に対して特に影響があると思われる以下の 3 項目を実機上で検証しました。

1. データ暗号化による性能影響に対する検証
2. PostgreSQL9.5 で導入された行単位のアクセス制御による実行計画への影響の検証
3. サーバログへの監査データ出力による性能および運用性の検証

5.2. 透過的暗号化

5.2.1. 検証ポイント

暗号化の負荷の影響を検証するため、以下のパターンで処理時間の相対比を取得しました。

表 5.1: 検証パターン(テーブル)

No.	テーブル	トリガー	ビュー	内容
1	平文	なし	なし	通常の平文テーブル。比較の基準となるテーブル。
2	pgcrypto 暗号化(1 列)	あり	あり	pgcrypto による暗号化テーブル。 トリガーにより暗黙的に暗号化。
3	pgcrypto 暗号化(15 列)			
4	キャスト(1 列)	あり	あり	トリガーによるキャストのみで暗号化は行なわない。 トリガーの負荷と暗号化の負荷の切り分け用。
5	キャスト(15 列)			
6	TDE 暗号化(1列)	なし	なし	Transparent Data Encryption for PostgreSQL を使用した透過的暗号化テーブル。本検証の主眼。
7	TDE 暗号化(15 列)			

以下の処理を検証します。

表 5.2: 検証パターン(処理)

No.	検証処理	内容
1	データロード	テキストデータを COPY FROM 文にてロードに要する時間を比較 pgcrypto 暗号化はトリガーにて実装
2	INSERT - SELECT	平文テーブルから SELECT し、暗号化テーブルへ INSERT する時間を比較 pgcrypto 暗号化はトリガーにて実装
3	SELECT	暗号化テーブルから復号化の時間を比較

暗号化のネックを探るため、実行時の CPU 負荷率および Disk I/O の情報を取得しました。
今回の検証環境は以下の通りです。クライアントとサーバのマシンは分けておらず同マシン上で実行しています。

表 5.3: 検証環境

項目		内容
OS		Red Hat Enterprise Linux Server release 6.5 (Santiago)
PostgreSQL のバージョン		PostgreSQL 9.5.0
サーバスペック	CPU	Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz 16Core
	メモリ	64GB

表 5.4: PostgreSQL パラメータ

パラメータ	設定値	備考
shared_buffers	10GB	対象データが載り切るサイズを確保

5.2.2. 検証データ

検証には、日本郵便株式会社が公開している郵便番号データを使用しました。[36]
公共性および日本語データである事を評価しました。
暗号化テーブルでは全列を対象としたものと、1列（郵便番号(7桁)）のみのと各2種類を用意しました。

データ件数　　：約 680 万件
テーブルサイズ　：約 1,100 MB
レコード長（平均）：約 118 バイト　　郵便番号(7 桁）は、レコード全体の 5.9%

表 5.5: 郵便番号データ詳細

内容	データ型	サンプル・データ
全国地方公共団体コード（JIS X0401、X0402）	半角数字	01101
（旧）郵便番号（5 桁）	半角数字	"060 "
郵便番号（7 桁）	半角数字	"0600042"
都道府県名	半角カタカナ	"ﾎｯｶｲﾄﾞｳ"
市区町村名	半角カタカナ	"ｻｯﾎﾟﾛｼﾁｭｳｵｳｸ"
町域名	半角カタカナ	ｵｵﾄﾞｵﾘﾆｼ(1-19 ﾁｮｳﾒ)
都道府県名	漢字	"北海道"
市区町村名	漢字	"札幌市中央区"
町域名	漢字	"大通西（1～19丁目）"
一町域が二以上の郵便番号で表される場合の表示	半角数字 (0 ,1)	1
小字毎に番地が起番されている町域の表示	半角数字 (0 ,1)	0
丁目を有する町域の場合の表示	半角数字 (0 ,1)	1
一つの郵便番号で二以上の町域を表す場合の表示	半角数字 (0,1)	0
更新の表示	半角数字 (0 ,1,2)	0
変更理由	半角数字 (0,1,2,3,4,5,6)	0

合計の平均長は118バイトです。

以下に本検証で使用するテーブル、トリガー、ビューの定義を記載します。

■平文（非暗号化）テーブル
通常のデータ型(char や text)で定義します。

```
CREATE TABLE zip(
    addr_code    char(5),
    zip_old      char(5),
    zip_code     char(7),
    ken_kana     text,
    shi_kana     text,
    cyo_kana     text,
    ken_name     text,
    shi_name     text,
    cyo_name     text,
```

36 http://www.post.japanpost.jp/zipcode/download.html

```
    mcyoiki_flg  char(1),
    koji_flg     char(1),
    cyome_flg    char(1),
    mzip_flg     char(1),
    update_flg   char(1),
    riyu_code    char(1)
);
```

■pgcrypto 暗号化テーブル

```
--- 1 列(zip_code)を bytea で定義

CREATE TABLE zip_aes1(
    addr_code    char(5),
    zip_old      char(5),
    zip_code     bytea,

     ～  中略  ～

    mzip_flg     char(1),
    update_flg   char(1),
    riyu_code    char(1)
);
```

```
---15 列を bytea で定義

CREATE TABLE zip_aes15(
    addr_code    bytea,
    zip_old      bytea,
    zip_code     bytea,

     ～  中略  ～

    mzip_flg     bytea,
    update_flg   bytea,
    riyu_code    bytea
);
```

■キャストテーブル　(pgcrypto 暗号化テーブルと同様)

```
--- 1 列(zip_code)を bytea で定義

CREATE TABLE zip_cast1(
    addr_code    char(5),
    zip_old      char(5),
    zip_code     bytea,

     ～  中略  ～

    mzip_flg     char(1),
    update_flg   char(1),
    riyu_code    char(1)
);
```

```
---15 列を bytea で定義

CREATE TABLE zip_cast15(
    addr_code    bytea,
    zip_old      bytea,
    zip_code     bytea,

     ～  中略  ～

    mzip_flg     bytea,
    update_flg   bytea,
    riyu_code    bytea
);
```

■TDE 暗号化テーブル

全項目を encrypt_text 型で定義します。

```
--- 1 列(zip_code)を encrypt_text で定義

CREATE TABLE zip_tde_aes1(
    addr_code    char(5),
    zip_old      char(5),
    zip_code     encrypt_text,

     ～  中略  ～

    mzip_flg     char(1),
    update_flg   char(1),
    riyu_code    char(1)
);
```

```
---15 列を encrypt_text 型で定義

CREATE TABLE zip_tde_aes15(
    addr_code    encrypt_text,
    zip_old      encrypt_text,
    zip_code     encrypt_text,

    ～  中略  ～

    mzip_flg     encrypt_text,
    update_flg   encrypt_text,
    riyu_code    encrypt_text
);
```

■pgcrypto 暗号化テーブル用のトリガ (15 列を暗号化した場合)

```
-----トリガー関数

CREATE or REPLACE FUNCTION zip_encrypto_tfunc15() RETURNS trigger AS $$
BEGIN
    NEW.addr_code   := encrypt(NEW.addr_code, 'zip'::bytea, 'aes');
    NEW.zip_old     := encrypt(NEW.zip_old , 'zip'::bytea, 'aes');
    NEW.zip_code    := encrypt(NEW.zip_code, 'zip'::bytea, 'aes');
```

```
      ～  中略  ～

    NEW.mzip_flg    := encrypt(NEW.mzip_flg, 'zip'::bytea, 'aes');
    NEW.update_flg  := encrypt(NEW.update_flg, 'zip'::bytea, 'aes');
    NEW.riyu_code   := encrypt(NEW.riyu_code, 'zip'::bytea, 'aes');

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

----- トリガー定義

CREATE TRIGGER zip_encrypt_trg15
    BEFORE INSERT ON zip_aes15
    FOR EACH ROW EXECUTE PROCEDURE zip_encrypto_tfunc15();
```

■キャストテーブル用トリガ (15 列を暗号化した場合)

```
----- トリガー関数

CREATE or REPLACE FUNCTION zip_cast_tfunc15() RETURNS trigger AS $$
BEGIN
    NEW.addr_code   := NEW.addr_code   ::bytea;
    NEW.zip_old     := NEW.zip_old     ::bytea;
    NEW.zip_code    := NEW.zip_code    ::bytea;

      ～  中略  ～

    NEW.mzip_flg    := NEW.mzip_flg    ::bytea;
    NEW.update_flg  := NEW.update_flg  ::bytea;
    NEW.riyu_code   := NEW.riyu_code   ::bytea;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;


----- トリガー定義

CREATE TRIGGER zip_cast_trg15
    BEFORE INSERT ON zip_cast15
    FOR EACH ROW EXECUTE PROCEDURE zip_cast_tfunc15();
```

■pgcrypto 暗号化テーブル用ビュー

参照用を容易にするためのビュー (15 列を暗号化した場合)

```
CREATE OR REPLACE VIEW zip_aes15_view AS
SELECT  convert_from(decrypt(addr_code, 'zip'::bytea, 'aes'),'UTF8') AS addr_code,
        convert_from(decrypt(zip_old, 'zip'::bytea, 'aes'),'UTF8') AS zip_old,
        convert_from(decrypt(zip_code, 'zip'::bytea, 'aes'),'UTF8') AS zip_code,

            ～  中略  ～

        convert_from(decrypt(mzip_flg, 'zip'::bytea, 'aes'),'UTF8') AS mzip_flg,
        convert_from(decrypt(update_flg, 'zip'::bytea, 'aes'),'UTF8') AS update_flg,
        convert_from(decrypt(riyu_code, 'zip'::bytea, 'aes'),'UTF8') AS riyu_code
FROM    zip_aes15;
```

■キャストテーブル用ビュー

参照用を容易にするためのビュー (15 列を暗号化した場合)

```
CREATE OR REPLACE VIEW zip_cast15_view AS
SELECT  convert_from(addr_code,'UTF8') AS addr_code,
        convert_from(zip_old,'UTF8') AS zip_old,
        convert_from(zip_code,'UTF8') AS zip_code,

            ～  中略  ～

        convert_from(mzip_flg,'UTF8') AS mzip_flg,
        convert_from(update_flg,'UTF8') AS update_flg,
```

```
        convert_from(riyu_code,'UTF8') AS riyu_code
FROM    zip_cast15;
```

5.2.3. 検証1（データロード）

COPY FROM 文によるデータロードの検証結果です。
各パターンによる所要時間を、絶対値および平文の所要時間を 1.0 とした相対値として記載します。
同様にテーブルサイズも絶対値および相対値で記載します。

表 5.6: 検証1の結果

No.	テーブル	所要時間(秒)		サイズ(MB)		備考
		絶対値	相対値	絶対値	相対値	
1	平文	29.07	1.00	1,099	1.00	基準値
2	pgcrypto 暗号化（1 列）	106.91	3.68	1,158	1.05	pgcrypto の負荷 トリガーのオーバーヘッド含む
3	pgcrypto 暗号化（15 列）	511.04	17.58	2,196	2.00	
4	キャスト(1 列）	85.69	2.95	1,099	1.00	キャストの負荷 トリガーのオーバーヘッド含む
5	キャスト(15 列）	218.40	7.51	1,099	1.00	
6	TDE 暗号化（1列）	41.53	1.43	1,172	1.07	透過的暗号化の負荷
7	TDE 暗号化（15 列）	153.13	5.27	2,414	2.20	

■所要時間

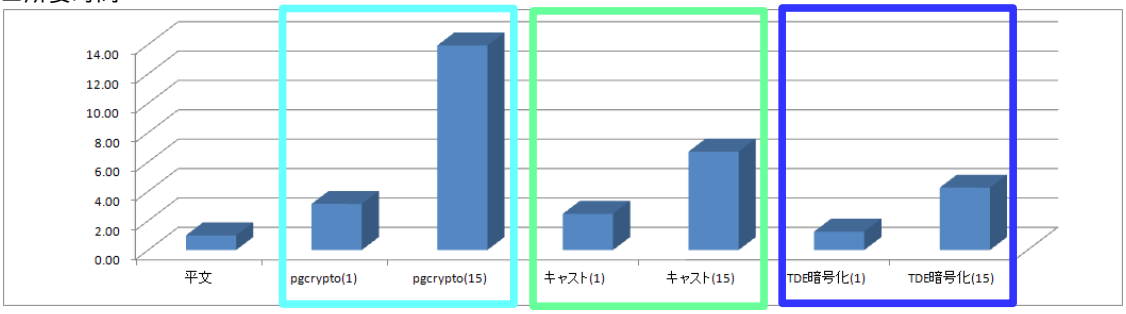


図 5.1: 検証1の結果(所要時間)

各方式において、1列と15列とで比較すると以下のようになります。
　pgcrypto　　：　20.9%
　キャスト　　：　39.2%
　TDE　　：　27.1%
単純にサイズや列数に比例するものではないことがわかります。

以下の関係を想定していました。
pgcrypto 暗号化の所要時間　≒　キャストオンリーの所要時間 ＋ TDE 暗号化の所要時間
実際には 以下の様に、ある程度想定に近い数値となりましたが、誤差も含んでおり完全には一致しません。
　1 列の場合　3.68 対 4.38
　15 列の場合　17.58 対 12.78

■テーブルサイズ

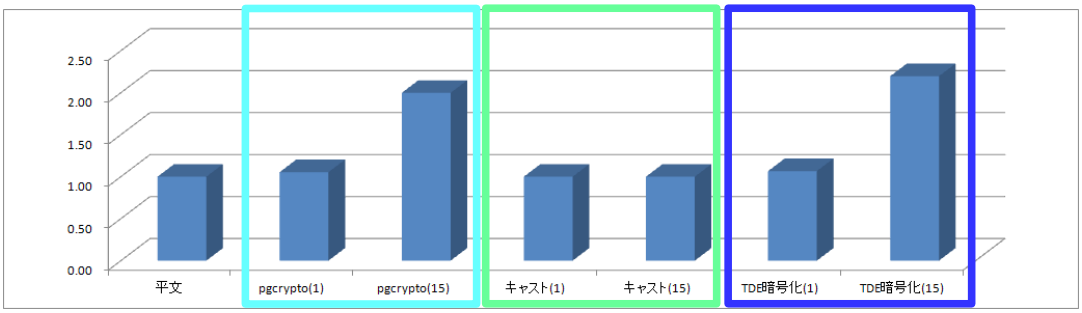


図 5.2: 検証1の結果(サイズ)

平文との比較としては、最大でも 2.2 倍となっています。

各方式において、1列と15列とで比較すると以下のようになります。

pgcrypto	:	52.7%
キャスト	:	100.0%
TDE	:	48.6%

pgcrypto と TDE では、暗号化の桁数が異なるため、TDE の方が大きくなります。
キャストは暗号化を行っていないため、同一サイズです。

以降の検証では、サイズは同一であるため、記載しておりません。

5.2.4. 検証 2 (INSERT – SELECT)

各パターンの INSERT – SELECT 文を記載します。

■平文　　zip テーブルから zip2 テーブルへ

```
insert into zip2 select * from zip;
```

■pgcrypto 暗号化テーブル　　zip テーブルから zip_aes2 テーブルへ
　SELECT 文で暗号化関数を実装しているため、トリガーが不要 (zip_aes2 テーブルにはトリガー未実装)

```
INSERT INTO zip_aes15_2
SELECT  encrypt(convert_to(addr_code,'UTF8'), 'zip'::bytea, 'aes') AS addr_code,
        encrypt(convert_to(zip_old,'UTF8'), 'zip'::bytea, 'aes') AS zip_old,
        encrypt(convert_to(zip_code,'UTF8'), 'zip'::bytea, 'aes') AS zip_code,

          ～　中略　～

        encrypt(convert_to(mzip_flg,'UTF8'), 'zip'::bytea, 'aes') AS mzip_flg,
        encrypt(convert_to(update_flg,'UTF8'), 'zip'::bytea, 'aes') AS update_flg,
        encrypt(convert_to(riyu_code,'UTF8'), 'zip'::bytea, 'aes') AS riyu_code
FROM    zip;
```

■キャストテーブル　　zip テーブルから zip_cast2 テーブルへ
　SELECT 文でキャストを実装しているため、トリガーが不要 (zip_cast2 テーブルにはトリガー未実装)

```
INSERT INTO zip_cast15_2
SELECT  addr_code::bytea AS addr_code,
        zip_old::bytea AS zip_old,
        zip_code::bytea AS zip_code,
        ken_kana::bytea AS ken_kana,

          ～　中略　～

        mzip_flg::bytea AS mzip_flg,
        update_flg::bytea AS update_flg,
        riyu_code::bytea AS riyu_code
FROM    zip;
```

■TDE 暗号化テーブル　　zip テーブルから zip_tde_aes2 テーブルへ

```
INSERT INTO zip_tde_aes15_2
SELECT  addr_code,
        zip_old,
        zip_code,

        ～　中略　～

        mzip_flg,
        update_flg,
        riyu_code
FROM    zip;
```

各パターンの所要時間を、平文の所要時間を 1.0 とした相対値として記載します。

表 5.7: 検証2の結果

No.	テーブル	所要時間(秒)		備考
		絶対値	相対値	
1	平文	20.66	1.00	基準値
2	pgcrypto 暗号化（1 列）	46.87	2.27	pgcrypto の負荷 トリガーのオーバーヘッドはない
3	pgcrypto 暗号化（15 列）	223.76	10.83	
4	キャスト(1 列)	23.66	1.15	キャストの負荷 トリガーのオーバーヘッドはない
5	キャスト(15 列)	36.54	1.77	
6	TDE 暗号化（1列）	38.08	1.84	透過的暗号化の負荷
7	TDE 暗号化（15 列）	206.50	10.00	

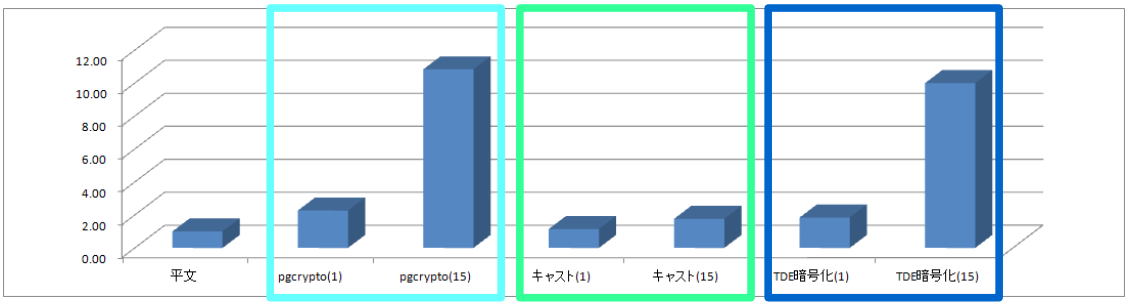


図 5.3: 検証 2 の結果

各方式において、1列と15列とで比較すると以下のようになります。

pgcrypto　：20.9%
キャスト　：65.8%
TDE　：18.4%

単純にサイズや列数に比例するものではないことがわかります。

pgcrypto の処理がトリガーを回避している事から、TDE とほぼ同じ所要時間となりました。

検証1に比較して相対速度が速いのは、以下の原因が考えられます。
1. ディスクアクセスが WAL 書き込みにより発生していることも原因として考えられます。

5.2.5. 検証3 (SELECT)

各パターンの SELECT 文を記載します。

```
平文                    : SELECT * FROM zip;
pgcrypto 暗号化(1)      : SELECT * FROM zip_aes1_view;      --- 復号化ビュー
pgcrypto 暗号化(15)     : SELECT * FROM zip_aes15_view;     --- 復号化ビュー
キャスト(1)             : SELECT * FROM zip_cast1_view;     --- 復号化ビュー
キャスト(15)            : SELECT * FROM zip_cast15_view;    --- 復号化ビュー
TDE 暗号化(1)           : SELECT * FROM zip_tde_aes1;
TDE 暗号化(15)          : SELECT * FROM zip_tde_aes15;
```

(例) TDE 暗号化(1)の場合の処理
所要時間の取得は psql の¥timing 機能にて行っているため、データをファイルに出力する時間は含まれません。

```
¥timing on
¥o select_tde_aes1.log

SELECT cipher_key_disable_log();
SELECT pgtde_begin_session('XXXXX');

SELECT * FROM pg_prewarm('zip_tde_aes1');
SELECT * FROM zip_tde_aes1;

¥q
```

各パターンによる所要時間を、平文の所要時間を 1.0 とした相対値として記載します。

表 5.8: 検証3の結果

No.	テーブル	所要時間(秒)		備考
		絶対値	相対値	
1	平文	12.50	1.00	基準値
2	pgcrypto 暗号化(1 列)	28.22	2.26	復号化の負荷(復号処理はビューで実装)
3	pgcrypto 暗号化(15 列)	217.97	17.44	
4	キャスト(1 列)	15.04	1.20	キャストの負荷(キャスト処理はビューで実装)
5	キャスト(15 列)	36.00	2.88	
6	TDE 暗号化(1列)	26.31	2.11	透過的復号化の負荷
7	TDE 暗号化(15列)	143.53	11.48	

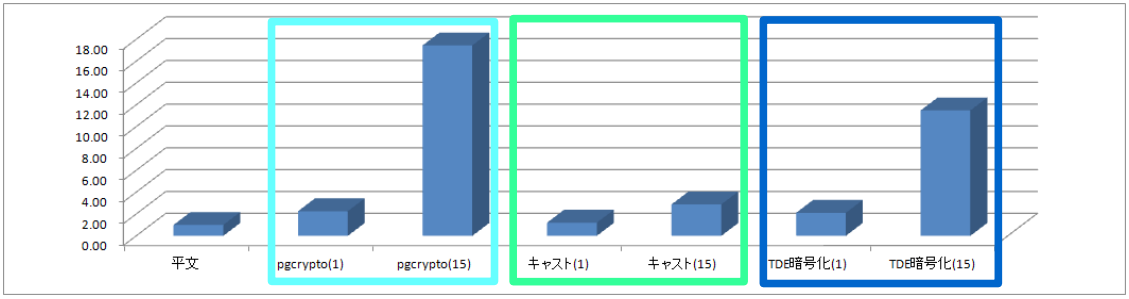


図 5.4: 検証3の結果

各方式において、1列と15列とで比較すると以下のようになります。

pgcrypto : 12.9 %
キャスト : 41.8 %
TDE : 18.3 %

単純にサイズや列数に比例するものではないことがわかります。

補足:
TDE 暗号化テーブルにて単純な SELECT を EXPLAIN ANALYZE で実行した場合、復号処理は演算されません。所要時間の検証の際にはご注意下さい。

5.2.6. CPU 負荷と DISK I/O

データロード時の CPU 負荷と DISK I/O を確認します。

■CPU 負荷
左から、TDE,キャスト、pgcrypto の状況です。
何れも特定の CPU の Wait% が0で 、User% および Sys% にて 100%となっています。

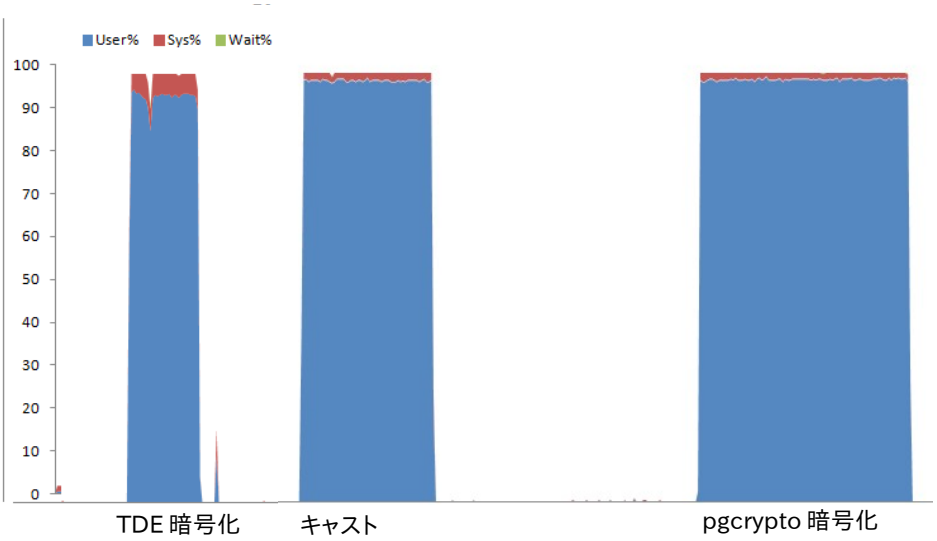


図 5.5: データロード時の CPU 負荷

■ディスク I/O
同時間帯の DISK I/O 量は以下です。何れも断続的であり、余裕があります。

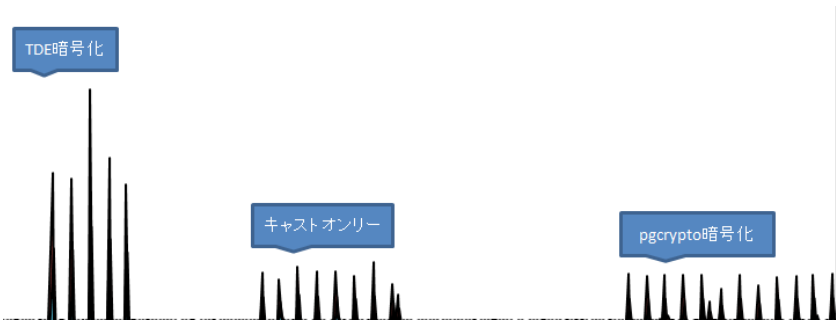


図 5.6: データロード時のディスク I/O

5.2.7. 考察

TDE による暗号化はアプリケーションへの適用は容易ですが、pgcrypto をベースにしているため、相応のオーバーヘッドはかかります。暗号化対象列の選定については、pgcrypto の場合と同様に慎重に検討し、オーバーヘッドを検証する事をお奨めします。

ディスク I/O より CPU 負荷が高いため、高性能な CPU を調達する事で、オーバーヘッドを低減する事が期待できます。

今回はデータのロードおよびデータの一括取得という DWH 系の処理を対象とした検証でした。そのため平文との差が顕著となりましたが、OLTP 系の処理であれば、目立った相違はない事が考えられます。用途や目的に応じてご検討下さい。

5.3. 行単位のアクセス制御

5.3.1. 検証ポイント

行単位セキュリティにてポリシーを設定している場合は、暗黙的にポリシー条件が任意の SQL に付与されます。ポリシーで指定した列に対してインデックスが付与されていた場合および付与されていない場合の実行計画に対する影響を検証します。

5.3.2. 検証データ

検証には、TDE と同じく、日本郵便株式会社が公開している郵便番号データを使用しました。[37]
ただしデータ件数は膨らませています(zip テーブル自身に INSERT ～SELECT を 3 回繰り返して 8 倍に)。

データ件数　　：約 5,400 万件
テーブルサイズ　：約 8,800 MB

今回の検証環境は以下の通りです。クライアントとサーバのマシンは分けておらず同マシン上で実行しています。

表 5.9: 検証環境

項目		内容
OS		Red Hat Enterprise Linux Server release 6.5 (Santiago)
PostgreSQL のバージョン		PostgreSQL 9.5.0
サーバスペック	CPU	Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz 16Core
	メモリ	64GB

表 5.10: PostgreSQL パラメータ

パラメータ	設定値	備考
shared_buffers	10GB	対象データが載り切るサイズを確保

5.3.3. 検証方法

1.ユーザ作成
非スーパーユーザの user_a および user_b を作成し、テーブル zip に対する SELECT 権限を付与します。
Bypass RLS 属性は付与しません。

(作成例)　user_a の場合(user_b も同様)

```
=# CREATE USER user_a LOGIN PASSWORD 'xxxxx';
=# GRANT select ON zip TO user_a;
=# ¥du
                                   ロール一覧
 ロール名 |                            属性                             | メンバー
----------+-------------------------------------------------------------+----------
 postgres | スーパーユーザ, ロールを作成できる, DB を作成でき           | {}
 user_a   |                                                             | {}
 user_b   |                                                             | {}
```

2. RLS ポリシー作成

37 http://www.post.japanpost.jp/zipcode/download.html

user_a,user_b にそれぞれ zip テーブルに対する RLS ポリシーを作成します。
　user_a の条件: addr_code like '13%' （東京都）
　user_b の条件: ken_name = '東京都'
addr_code 列にはインデックスが付与され、ken_name 列には付与されていないという差異があります。

（RLS ポリシー作成例: user_a）

```
CREATE POLICY user_a_zip_select ON zip
  FOR SELECT
  TO user_a
  USING (addr_code like '13%');
```

（RLS ポリシー作成例: user_b）

```
CREATE POLICY user_b_zip_select ON zip
  FOR SELECT
  TO user_b
  USING (ken_name = '東京都');
```

3. SQL 実行

以下のパターンにて実行計画および処理時間を確認します。

表 5.11: 検証パターン

ケース	実行ユーザ	RLS ポリシーによる暗黙条件	SQL 文で明示的に指定した Filter 条件
ケース①	postgres	RLS ポリシーなし	なし
ケース②	user_a	RLS ポリシーあり（addr_code）	なし
ケース③	user_b	RLS ポリシーあり（ken_name）	なし
ケース④	user_b	RLS ポリシーあり（ken_name）	あり（addr_code）
ケース⑤	postgres	RLS ポリシーなし	あり（addr_code および ken_name）

各ケースの実行SQL（これを EXPLAIN (ANALYZE,BUFFERS) にて実行）

```
ケース①:  SELECT * FROM zip;
ケース②:  SELECT * FROM zip;
ケース③:  SELECT * FROM zip;
ケース④:  SELECT * FROM zip WHERE addr_code LIKE '13%';
ケース⑤:  SELECT * FROM zip WHERE addr_code LIKE '13%' AND ken_name = '東京'
```

ケース④とケース⑤は同じ結果を返すSQLとなります。

なお所要時間取得に際しては以下の条件とし、ぶれを少なくしています。

- zip テーブルを pg_prewarm に事前にロードする事で、ヒット率を 100%としました（共有バッファは zip テーブルに対して十分なサイズを用意しました）。
- 各ケースとも 5 回実行し、中央値をそのケースの値としました。

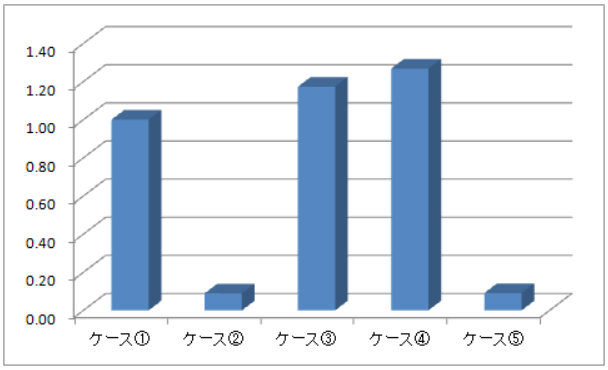
```
■共有バッファのサイズ

=# show shared_buffers;
 shared_buffers
----------------
 10GB

■zip テーブルのサイズ

=# SELECT pg_size_pretty(pg_relation_size('zip'));
 pg_size_pretty
----------------
 8770 MB

■zip テーブルの事前ロード

=# SELECT * FROM pg_prewarm('zip');
 pg_prewarm
------------
    1122500
(1 行)

■バッファの確認（pg_buffercache エクステンションが必要）

=# SELECT count(*) FROM pg_buffercache
-# WHERE  relfilenode=(SELECT relfilenode
(#                     FROM   pg_class
(#                     WHERE  relname = 'zip');
  count
---------
 1122500
(1 行)
```

5.3.4. 検証結果

所要時間はケース① を 1.00 とした場合の相対比です。
網掛け行はインデックス・スキャンとなったケースです。

表 5.12: 検証結果

ケース	アクセスパス	所要時間（相対比）	備考
ケース①	シーケンシャル・スキャン	1.00	単純なフルテーブル走査であり、ベースとなるケース
ケース②	インデックス・スキャン	0.09	RLS ポリシーの索引が使用された
ケース③	シーケンシャル・スキャン	1.17	索引となる条件を指定してないため、フルテーブル走査
ケース④	シーケンシャル・スキャン	1.27	明示的条件の索引が使用されない
ケース⑤	インデックス・スキャン	0.09	明示的条件の索引が使用された



■ケース①の実行計画　フルテーブル走査

```
=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM zip;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------
 Seq Scan on zip  (cost=0.00..1667501.60 rows=54500160 width=133) (actual time=
   0.009..8486.966 rows=54500160 loops=1)
   Buffers: shared hit=1122500
 Planning time: 1.008 ms
 Execution time: 12439.443 ms
(4 行)
```

■ケース②の実行計画　インデックス・スキャン (bitmap index scan)

```
=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM zip;
-----------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on zip  (cost=35424.00..1233305.28 rows=1668109 width=133) (actual
time=341.260..979.765 rows=1675520 loops=1)
   Filter: (addr_code ~~ '13%'::text)
   Heap Blocks: exact=43171
   Buffers: shared hit=43171 read=4582
   ->  Bitmap Index Scan on zip_addr_code  (cost=0.00..35006.98 rows=1669441 width=0)
         (actual time=328.654..328.654 rows=1675520 loops=1)
         Index Cond: ((addr_code >= '13'::bpchar) AND (addr_code < '14'::bpchar))
         Buffers: shared read=4582
 Planning time: 7.791 ms
 Execution time: 1088.510 ms
(9 行)
```

■ケース③の実行計画　フルテーブル走査

```
=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM zip;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------
 Seq Scan on zip  (cost=0.00..1803752.00 rows=1685872 width=133) (actual time=
   21.751..15377.808 rows=1675520 loops=1)
   Filter: (ken_name = '東京都'::text)
   Rows Removed by Filter: 52824640
   Buffers: shared hit=1122500
 Planning time: 1.417 ms
 Execution time: 15478.017 ms
(6 行)
```

■ケース④の実行計画　シーケンシャル・スキャン

```
=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM zip WHERE addr_code LIKE '13%';
                                                              QUERY PLAN
------------------------------------------------------------------------------------------------------
 Subquery Scan on zip  (cost=0.00..1824825.40 rows=8429 width=133) (actual time=
   17.067..13725.720 rows=1675520 loops=1)
   Filter: (zip.addr_code ~~ '13%'::text)
   Buffers: shared hit=1122500
   ->  Seq Scan on zip zip_1  (cost=0.00..1803752.00 rows=1685872 width=133) (actual time
       =17.059..13328.265 rows=1675520 loops=1)
         Filter: (ken_name = '東京都'::text)
         Rows Removed by Filter: 52824640
         Buffers: shared hit=1122500
 Planning time: 0.239 ms
 Execution time: 13825.439 ms
(9 行)
```

RLSポリシーが優先処理され、SQL で直接指定した条件の部分がサブクエリーとなっています。

■ケース⑤の実行計画　インデックス・スキャン (bitmap index scan)

```
=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM zip WHERE addr_code LIKE '13%'
                                                AND   ken_name = '東京';
                                                              QUERY PLAN
------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on zip  (cost=35006.98..1237061.85 rows=1 width=133) (actual time=
   982.990..982.990 rows=0 loops=1)
   Filter: ((addr_code ~~ '13%'::text) AND (ken_name = '東京'::text))
   Rows Removed by Filter: 1675520
   Heap Blocks: exact=43171
   Buffers: shared hit=47753
   ->  Bitmap Index Scan on zip_addr_code  (cost=0.00..35006.98 rows=1669441 width=0)
         (actual time=264.904..264.904 rows=1675520 loops=1)
         Index Cond: ((addr_code >= '13'::bpchar) AND (addr_code < '14'::bpchar))
         Buffers: shared hit=4582
 Planning time: 0.412 ms
 Execution time: 983.056 ms
(10 行)
```

論理的に等価なケース④とケース⑤にて、実行計画が異なることは注目に値します。

5.3.5. 考察

SQL 文で明示的に指定した条件と RLS ポリシーでは、RLS ポリシーが優先処理され、明示的に指定した条件はサブクエリーの扱いとなっています。

上記の実行計画は本データの場合であり、プランナ統計情報などにより異なる可能性があります。
RLS ポリシーを作成する場合には、実行計画も意識して設計および検証を行う事をお奨めします。

5.4. PostgreSQL サーバログへの監査データ出力による性能および運用性の検証

5.4.1. 目的

PostgreSQL の標準機能を利用して監査情報を取得する場合、PostgreSQL サーバのパラメータである、log_statement を設定し発行された SQL 文を PostgreSQL のサーバログに出力し他データに対して、監査を実施することになります。PostgreSQL9.5 現在では log_statement の出力範囲の設定値は none,ddl,mod.all と 4 つの設定があります。これらのパラメータをデータベースのユーザに紐づけることでユーザ毎に出力データ量を切り替える方法はありますが、重要なデータのアクセスに制限するなど監査データの出力範囲の指定を柔軟に行うことができません。そのため、監査対象のデータやアクセス方法に対して確実な情報を残すためには[4.2 pgaudit 拡張モジュールの使用]でも記載されている通り、全ての SQL 文出力が必要となるケースが多くなることが想定されます。
この場合、すべての SQL 文を出力することで性能やデータ出力量による運用への影響に対する考慮が必要な可能性が考えられます。

そこで、今回は PostgreSQL 標準のベンチマークツールである pgbench を利用し、全ての SQL 文を出力する log_statement=all と SQL 文を出力しない log_statement=none の間で、参照系のクエリが大量に流れた場合の性能やログ出力量に対する影響について検証を行いました。

5.4.2. 検証環境

今回の検証環境は以下の通りです。クライアントとサーバのマシンは分けておらず同マシン上で実行しています。

表 5.13: 評価環境一覧

項目		内容
OS		Red Hat Enterprise Linux 6.5
PostgreSQL のバージョン		PostgreSQL 9.5 Beta2
評価マシン	CPU	インテル® Xeon® プロセッサー E5645 (6core) 2 Socket
	メモリ	24GB
	HDD	SATA 500GB(7200rpm)×4 台　RAID5

検証パターンとしては、データベースエンジンが発揮する性能による影響の違いを確認するため、以下の 2 パターンで実施することとしました。

- データベース(pgbench の scale factor=100 約 1.5GB) がオンメモリ上に格納され、SELECT の性能が非常に高いケース
- データベース(scale factor= 2000 約 30GB)はメモリのサイズより大きく、ディスクアクセスを伴うケース

なお性能関連の共通的なパラメータについては、マシンスペックにあわせて以下のみを変更しております。

```
maintenance_work_mem = 256MB
shared_buffers = 4096MB
```

pgbench の計測条件は以下です。

```
$ pgbench -S -c 48 -j 24 -P 60 -r -T 180
```

5.4.3. 検証結果

pgbench で準備したデータベースに対して、log_statement=none と log_statement=all の違いの計測を行っています。なお、計測は 3 回実施しています。

a) Scale factor =100 (DB サイズ=約 1.5GB)の場合

まずは、Scale Factor =100 のデータベースが全てバッファに乗った状態で測定した結果を説明します。3 回の実行測定結果は以下の表の通りとなりました。

表 5.14: SF =100 の測定結果（1測定あたり 3分間）

	log_statement=none	log_statement=all		
実行回数	TPS	TPS	ログファイルサイズ(kB)	
1	144912.3913	116192.21	3,155,539	
2	145149.3546	117703.08	3,204,668	
3	144898.4911	117786.57	3,206,972	性能比
平均	144986.75	117227.29	3,189,060	80.85%

このTPS性能の測定結果（平均値）をグラフにしたものが以下となります。

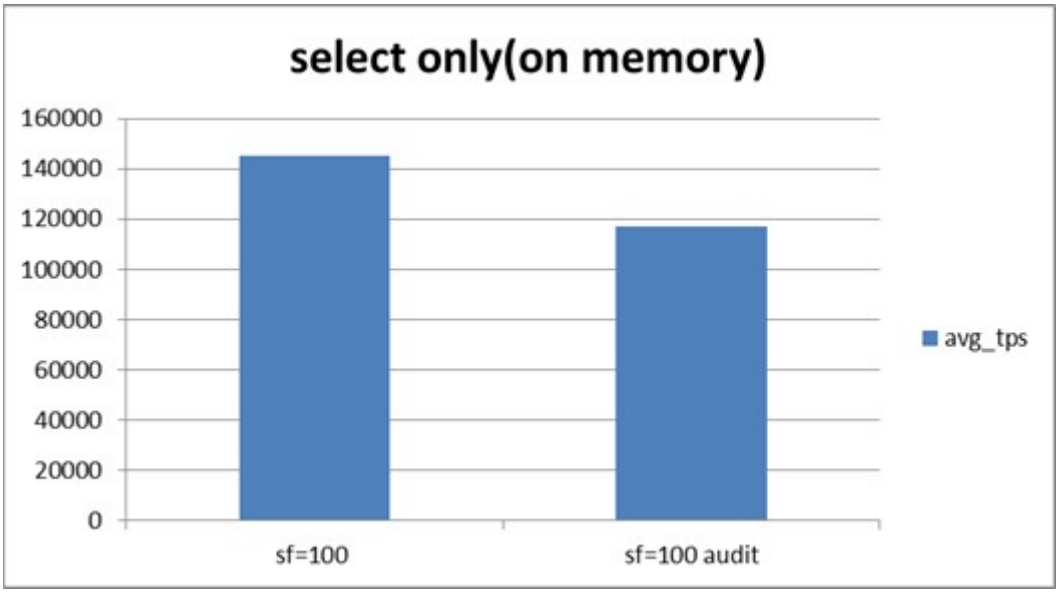


図 5.7: TPS 性能の結果(データベースがオンメモリの場合）

左が log_statement = none のSQLのログ出力を行っていないものの平均値、右側が log_statement=all で全てのSQL文を出力したものの平均値となっています。この結果では、log_statement =all で全SQLをログファイルに出力することで約 20%の性能低下がみられています。
ログ出力量については、1回あたり3分間の測定で約 3GB のログデータが出力されています。

計測中の sar からの CPU の稼働状況は以下となりました。

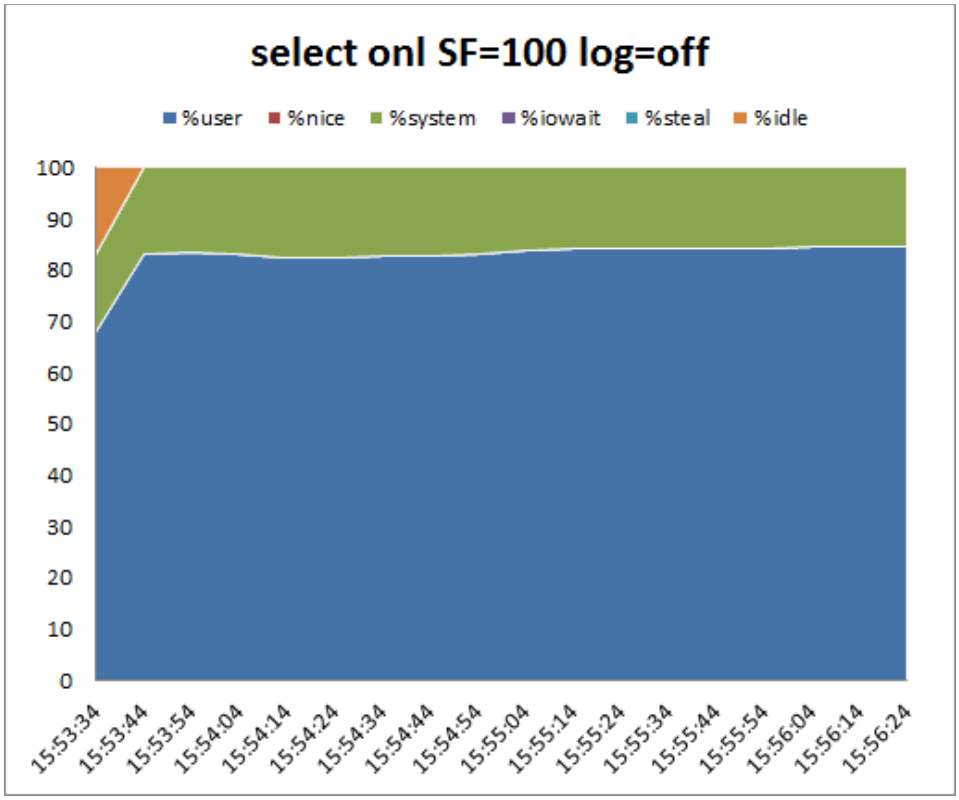


図 5.8: オンメモリでの PostgreSQL サーバの CPU 負荷状況（log_statement = none）

log_statement=none（SQL のログ出力なし）での sar の結果からはほぼ idle は出ておらず、PostgreSQL の検索処理でサーバの負荷を十分に掛けられていることがわかります。

一方で、log_statement=all（SQL のログ出力あり）の場合が次の結果となります。

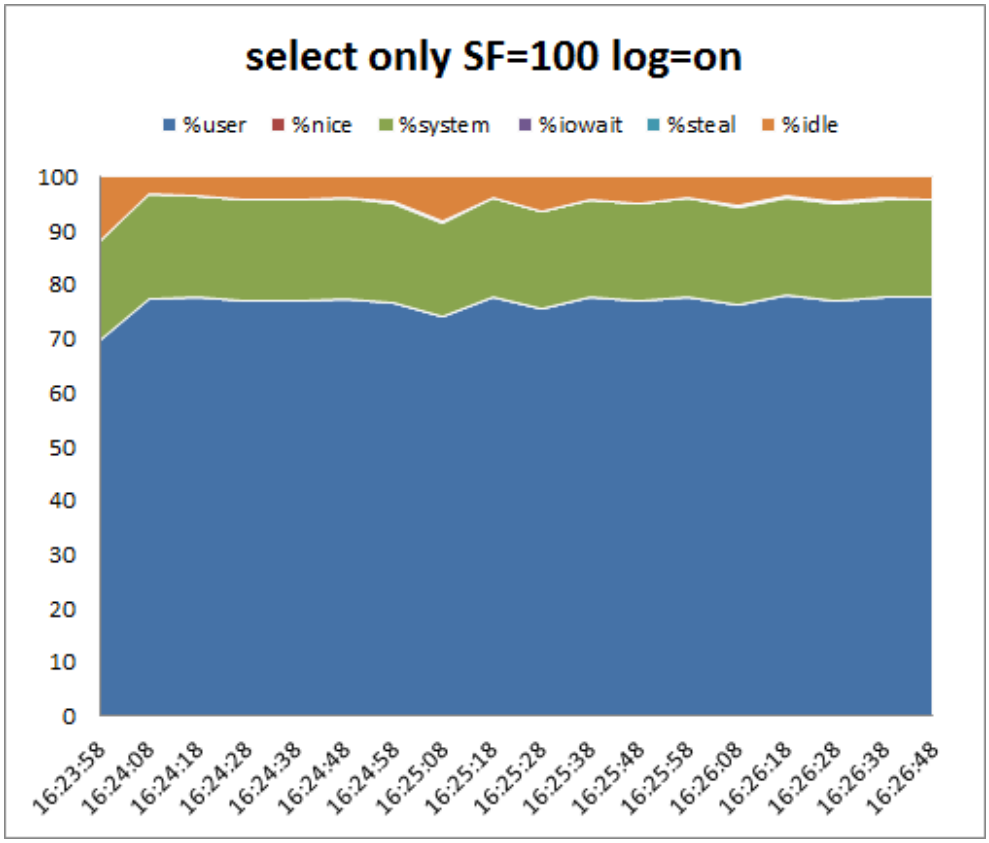


図 5.9: オンメモリでの PostgreSQL サーバの CPU 負荷状況（log_statement = all）

log_statement=all（SQL のログ出力あり）でも、sar の結果はほぼ同様の傾向ではありますが、ログデータ出力による影響かわずかながら%idle が出ておりパフォーマンスが劣化している結果により上記のような結果になったと推測されます。

b)　スケールファクター =2000　(DB サイズ＝約 30GB)の場合

次に、Scale Factor =2000 のデータベースはバッファに乗らない、ストレージのアクセスが必要な状況下で測定した結果を説明します。3 回の実行測定結果は以下の表の通りとなりました。この時、1 分間あたりのログデータの出力量は約13 MB 程度です。

表 5.15: SF =2000 の測定結果

	log_statement=none	log_statement=all		
実行回数	tps 結果	tps 結果	Log ファイルサイズ(kB)	
1	1356.78	1453.73	40,028	
2	1375.57	1351.34	37,192	
3	1385.12	1372.92	37,800	性能比
平均	1372.49	1392.66	38,340	101.47%

同様に TPS 性能の測定結果（平均値）をグラフにしたものが以下となります。

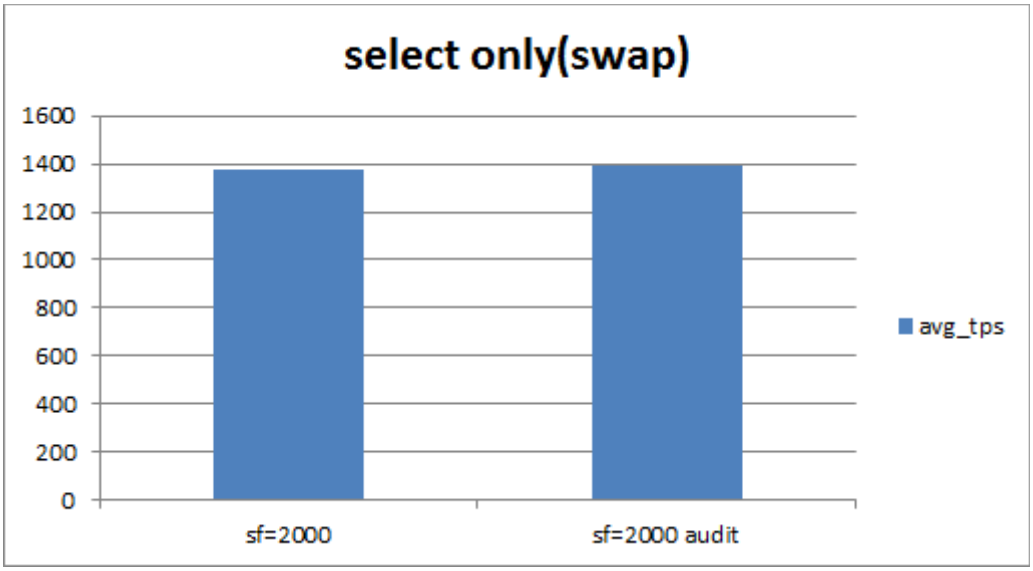


図 5.10: TPS 性能の結果(ストレージアクセスあり)

先ほどと同様、左が log_statement = none のSQLのログ出力を行っていないものの平均値、右側が log_statement=all で全てのSQL文を出力したものの平均値となっています。
ほとんど性能に変化はありませんが、わずかながらですが、ログ出力をしたときのほうが性能が良くなってしまいました。ただし、これは誤差レベルと考えています。

Scale factor=2000 での計測中の sar からの CPU の稼働状況は以下となりました。

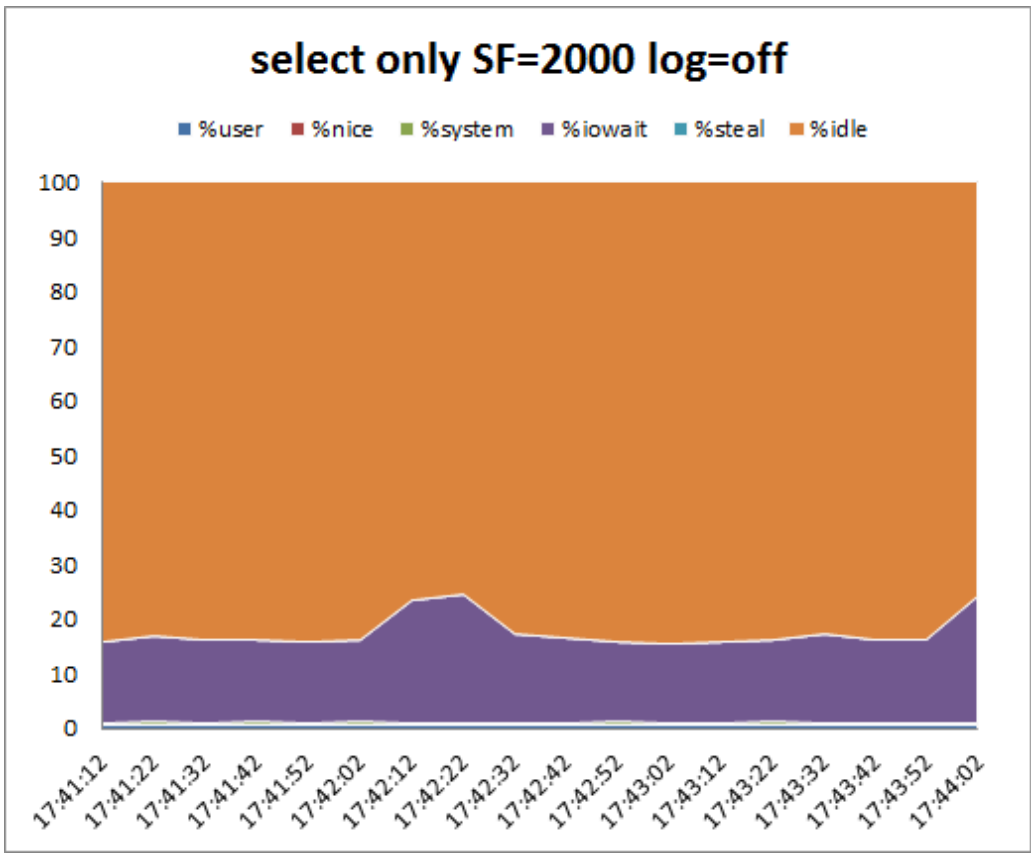


図 5.11: DB のストレージアクセス有り時の PostgreSQL サーバの CPU 負荷状況(log_statement = none)

この結果からわかる通り、データベースのディスクアクセスのため、データベースサーバへの CPU 負荷はほとんど上がっていないことが確認できます。次に log_statment =all の CPU 負荷状況を掲載していますが傾向は変わりませんでした。

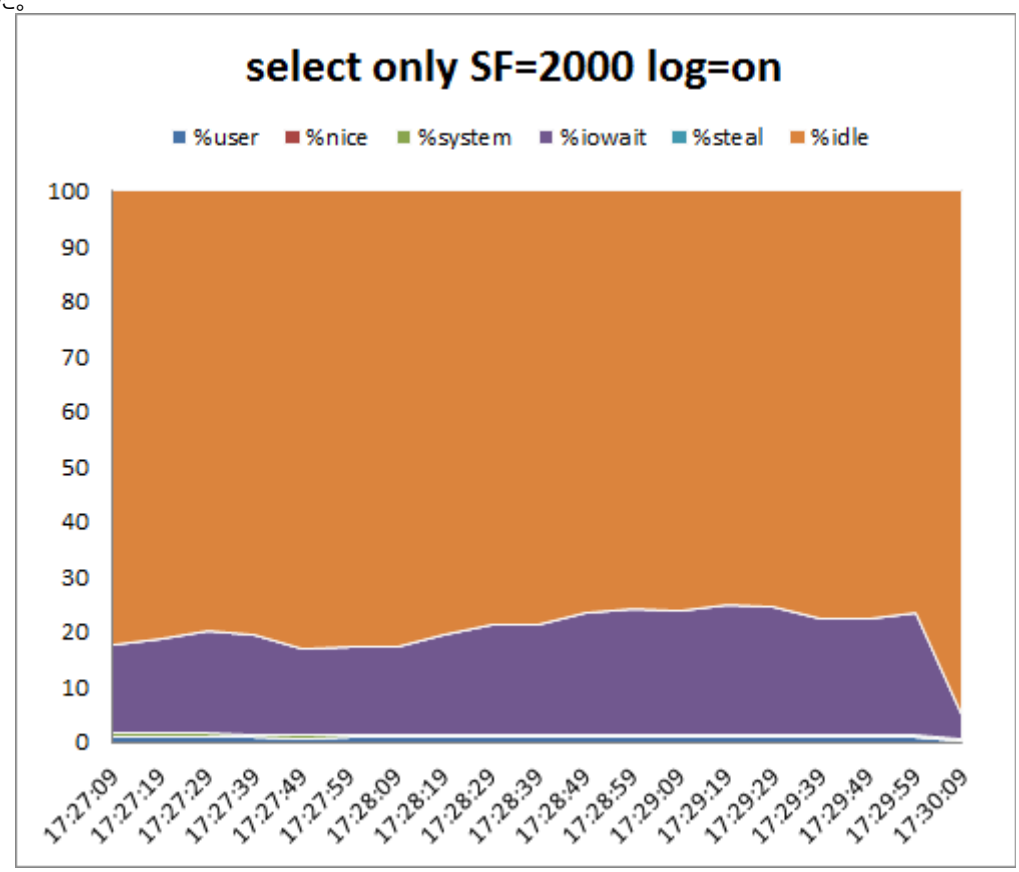


図 5.12: DB のストレージアクセス有り時の PostgreSQL サーバの CPU 負荷状況（log_statement = all）

5.4.4. 考察

この検証結果から、log_statement=all に設定した際に、ログデータへの書き込み処理に伴うクエリのパフォーマンスの影響は限定的であることが確認されました。特にバッファ中のデータに対する SELECT という高いパフォーマンスが得られる状況下において、監査データの書き込みを並行して実施するといった、相対的に性能が落ちそうな条件でも、約 19%の劣化に留まっています。データ量が大きくストレージへの負荷が挙がっているケースではパフォーマンスの低下は確認できませんでした。
この結果から監査データの出力によるデータベースエンジンへのパフォーマンスの影響は比較的軽微と言えるのではないかと考えています。

しかし、今回のケースではバッファ中に対する単純なSELECTを実行した場合には、1分あたりおおよそ 1GB のログデータが出力されています。これは非常に膨大なデータ量であると言えます。仮に今回評価したサーバで同様のクエリが継続して発生した場合には、1日あたり約 1.5TB、1か月では約 45TB/月のログデータが出力されることになります。
バッファ中ではなくストレージのアクセスが発生した場合でも、1 分あたり約 13MB のログデータが出力されました。この場合でも、1 日あたり約 18GB、1 か月では約 500～600GB、1 年間では約 6TB 程度のログデータが出力されます。
出力された監査データは、事後に分析を行うことやリアルタイムでアラートをあげることが求められます。上記のデータ量に対してのバックアップ取得や監査データの分析についての対応策も検討する必要があります。さらには、ログデータを格納するストレージが枯渇してしまえば、監査データが正しく出力されなくなるという可能性も含んでいます。

今回は、pgbench という非常に軽量なクエリが大量に発生するベンチマークツールを使っています。よって、すべての PostgreSQL サーバが同様のログデータの出力量になるとは限りません。しかし、今回は 2012 年頃のハードウェアを用いた検証であり、特別に高い性能のハードウェアを用いているというケースでもないことから、現実的にあり得るデータ量であると認識しています。実際に log_statement =all で監査ログ出力とする場合には、対象システムのログ出力量を事前に確認し、不正検知の方法や出力データの運用方法について十分な検討を行うことをお薦めします。

この検証結果から、監査の出力データは監査に求められる重要なテーブルや怪しいクエリなど、必要最小限に制限したものを出力する機構が重要であることが再確認できました。そのため、PostgreSQL では 2015 年現在コミュニティで議論されている pg_audit などの外部ツールによる監査の出力データの絞込み機能について有効であり、今後の発展が期待されます。

6. おわりに

本書の今年度版(2015 年度)では昨年度に引き続き、PCI DSS で定義されるセキュリティポリシーを PostgreSQL で扱うためのノウハウを解説すると同時に、昨年度の検討を更に掘り下げた成果を報告しました。今年度版は昨年度版から、以下の点の記述を追加しました。

- PostgreSQL 本体の行レベルセキュリティ、透過的暗号化モジュール(日本電気株式会社開発、OSS 版)など、コミュニティでのセキュリティ強化の動きを反映して記述を強化した
- また、これらの機能の紹介にあたって、実機上での検証を通じて性能面の特性を明らかにした
- パスワード関連、DB 監査関連の記述を強化した

これらの追加情報によって、本書がより皆様の業務の参考になればと願っております。

検討結果全体を見ると、PostgreSQL 単体では困難なものもありますが、OS の機能やサードパーティ/商用製品と組み合わせることで、セキュリティに関する課題を概ねクリアできることが確認できます。
なお、本書で提示している実行例等は最低限の機能確認や性能状況の確認を目的としているため、業務システムにおける性能面への影響や、実務面での要求を考慮したセキュリティ確保については、実際のシステムに適用するにあたって、実地に検討・検証することをお勧めします。

本書をご覧になり、ご興味を持たれた方、よりよい機能や設定をご指摘いただける方がいらっしゃいましたら、忌憚のないご意見お待ちしております。ぜひ PostgreSQL の普及促進をめざし、PostgreSQL エンタープライズ・コンソーシアムへの参加および活発な議論/検証を共に実施していくことをお願いいたします。

著者

版	所属企業・団体名	部署名	氏名
2014 年度 WG3 活動報告書 セキュリティ編 第 1 版	株式会社アシスト	データベース技術本部	柘植 丈彦
	株式会社アシスト	データベース技術本部	喜田 紘介
	SRA OSS, Inc. 日本支社	マーケティング部	高塚 遙
	NTT ソフトウェア株式会社	クラウド＆セキュリティ事業部	勝俣 智成
	NTT ソフトウェア株式会社	クラウド＆セキュリティ事業部	山本 育
	サイオステクノロジー株式会社	テクニカルサポート部	佐藤　仁
	大日本印刷株式会社	C&I 事業部プラットフォームサービス本部	亀山 潤一
	大日本印刷株式会社	C&I 事業部プラットフォームサービス本部	田中 良幸
	日本電気株式会社	クラウドプラットフォーム事業部	川畠 輝聖
	日本電気株式会社	クラウドプラットフォーム事業部	慶松 明嗣

版	所属企業・団体名	部署名	氏名
PostgreSQL セキュリティガイド (旧文書名：2014 年度 WG3 活動報告書) 第 2 版	株式会社アシスト	データベース技術本部	柘植 丈彦
	株式会社アシスト	データベース技術本部	喜田 紘介
	日本電信電話株式会社	OSS センタ	坂田 哲夫
	日本電気株式会社	クラウドプラットフォーム事業部	川畠 輝聖