SQLインジェクション総"習"編

第五回 関西DB勉強会
SQLインジェクションの全てを知る

エレクトロニック・サービス・イニシアチブ有限会社
大垣 靖男

自己紹介

- 氏名： 大垣　靖男

- SNS：yohgaki(FB/G+/TW)
- https://blog.ohgaki.net/
- yohgaki@ohgaki.net

- https://www.es-i.jp/

- エレクトロニック・サービス・イニシアチブ有限会社 代表取締役社長、PostgreSQLユーザー会 理事、PHP技術者認定 顧問、BOSSCON CTO、岡山大学大学院 非常勤講師
- Webシステム開発のコンサルティング、テクニカルサポート、セキュリティ検査など
- PHPコミッター

テーマ

「SQLインジェクション」の全てを俯瞰する

SQLインジェクションは「インジェクション」と「その対策」を知る良い題材です。

まず体系的な「セキュリティ対策」の定義から

ISO 27000のセキュリティ要素

- 機密性
- 完全性
- 可用性
- 信頼性
- 真正性
- 否認防止・責任追跡性

SQLインジェクション対策に於いても、SQLコマンドのインジェクションを防ぐ事だけ、がセキュリティ対策ではない

まず体系的な「セキュリティ対策」の定義から

- ISO 27000と「セキュリティ対策」
  - 「セキュリティ対策」という曖昧な用語は定義していない
  - セキュリティ対策とは**「リスク管理」（Risk Management）**だと考えられており、**「リスクに変化を与える全ての対応策」（Risk Treatment）**がセキュリティ対策だと考えられている
  - **リスクの削減だけがセキュリティ対策、はよくある間違い。**
  - **リスクの受け入れもセキュリティ対策（＝リスク管理策）として管理する**のが“**体系的なセキュリティ対策**”です。
- 詳しくは
  - https://blog.ohgaki.net/it-security-concept

リスク受容を
考慮・管理しないは
よくある間違い、
しかも致命的なので注意！

SQLインジェクションとは？

- SQL文に開発者が意図しない「**SQL文**」を挿入（インジェクション）して、「**不正なSQL操作を実行**」させる（**狭義**）

- SQL文に開発者が意図しない「**命令またはデータ**」を挿入（インジェクション）して、「**不正なデータ/操作を保存/実行**」させる（**広義**）

- 開発者にとって“**狭義**”、“**広義**”はあまり意味がない
  - **不正SQL実行攻撃防止“以外”の対策も同じように大切**
  - **DBMSを攻撃されることは同じ**
  - **どちらも同じ「インジェクション」**

攻撃者にとっては、攻撃方法が異なる、ので意味がある。

開発者にとっては、防御方法の基本が同じ、なのであまり意味がない。

まず結論！

- Q：SQLインジェクション対策は簡単か？
- A：とても簡単です！JavaScriptインジェクションに比べると万倍簡単に対応・修正できます。

- Q：簡単なのになぜ無くならないのか？
- A：問題の理解・対策方法が合理的ではないからです。

- Q：簡単に完全な対策を説明すると？
- A：必要十分条件を満す、と考えると論理的に正しい対策が解ります。

SQLインジェクションの基礎知識

典型的なSQLインジェクションの例

- SELECT ＊ FROM mytable WHERE id = $userid;

- $useridに“123; DROP TABLE mytable;”が入っている場合
  - 複数文実行をサポートするDBの場合、攻撃が簡単

- SELECT ＊ FROM mytable WHERE id = 123; DROP TABLE mytable;
  - テーブル削除をはじめ、何でも実行される可能性がある

典型的なSQLインジェクションの例

- SELECT ＊ FROM mytable WHERE id = $userid;
- $useridに“123 UNION SELECT * FROM othertable”などが入っている場合
- SELECT ＊ FROM mytable WHERE id = 123 UNION SELECT * FROM othertable
  - 他のテーブルのデータを盗まれる可能性がある

“間接”と“直接”SQLインジェクション

- “間接”インジェクションは“直接”脆弱なコードを攻撃するのではなく「**一旦、別の場所に保存した後、間接的にインジェクション攻撃を行う**」手法

基本的にデータは信頼できない
信頼済みの「場所」に保存されて
いるかどうかは無関係

信頼境界線： https://blog.ohgaki.net/how-to-draw-and-protect-trust-boundary

“間接”と“直接”SQLインジェクション

- “直接”SQLインジェクション対策を完全に行っていれば、“間接”SQLインジェクションは問題にならない
- 間接SQLインジェクションの例：
  - $usernameを保存時には正しくSQL文字リテラルとしてエスケープされているとする
  - $usernameに” attacker’; update users set password=‘hacker“ where username=‘admin”を保存
  - $usernameがDBに保存された信頼できるデータとしてエスケープ無しで次のクエリが実行されるとする。 この場合、次のように‘admin’のパスワードが’hacker’に更新される
  - “SELECT * FROM users WHERE username= ‘$username’;“は SELECT * FROM users WHERE username= ‘attacker’; update users set password=‘hacker’ where username=’admin’;“として実行される

どこに保存されようと外部入力は信頼できない！

数値型カラムのデータも安心できない　→　SQLiteのカラムは基本全て文字列型

“間接”と“直接”SQLインジェクション

- “間接”インジェクションが作られる原因は**信頼境界線の理解不足**
  - **システム設計コンテクストと**
    ソフトウェアコンテクストの信頼境界線は“異なる”
    → **ソフトウェアの信頼境界限界は**
    同一HW上の同一プロセス・スレッド上
    のコードまで
  - **信頼境界線内でもリスク要因が存在する**
    → **特に“データ”の盲目的信頼は絶対ダメ！**
    通常データは“外部”のモノ（ソフトウェアの信頼境界線の外）
- **真正性識別の過誤が原因のリクエストフォージェリ（詐称）と同類**
  - ログイン中のユーザーからの**リクエストが真正とは限らない**
  - DBに保存された**データが安全とは限らない**

よくある間違いなので
注意！！

信頼境界線： https://blog.ohgaki.net/how-to-draw-and-protect-trust-boundary

典型的な間違い

- 「SQLインジェクション対策には
  プリペアードクエリ・プレイスホルダ“だけ”を使えばよい」

- 「SQLインジェクション対策に
  エスケープは必要ない」

- 「SQLインジェクション対策に
  入力バリデーションは必要ない」

なぜ対策が簡単なSQLインジェクションが無くならないのか？

人は不合理な判断を簡単に行う生き物で、これは避けられない

問題対処が合理的にできない理由

- **手近な情報による誤謬**
  「人々はデータではなく、強い印象を受けたことや、最初に頭に思い浮かんだ事柄で物事を判断する」
- 「人間がリスクを正しく判断できない一つは過信であり、もう一つは将来の**あらゆる可能性を徹底して検証する能力の欠如**だ」

- **不合理 誰もがまぬがれない思考の罠100 ← お勧めです！**
  - スチュアート・サザーランド
  - http://amzn.to/2r8H0Ch

SQLインジェクションのパターン

- **コンテクストの異なる個所にインジェクション**
  - **それぞれのコンテクストは更に細かいコンテクストを持つ**

パースするデータの場合に共通のパターン！
SQL、HTML、JavaScriptなど

- **パラメーター**にインジェクション
- **識別子**にインジェクション
- **SQL語句**にインジェクション

典型的なSQLインジェクションだが、これが全てではない！

手近な情報による誤謬
・最初に思い浮かんだコト
・強い印象を受けたモノ
により、これが全てだと間違える誤謬

誤謬：一見正しくみえるが誤っている推理。推理の形式に違背したり，用いる言語の意義が曖昧（あいまい）であったり，推理の前提が不正確であることから生ずる。詭弁（きべん）。論過。虚偽。
http://www.weblio.jp/content/%E8%AA%A4%E8%AC%AC

「合成の誤謬」のように個別に見ると正しいことが、全体としては間違っている場合もある。正しいことの積み重ねは必ずしも全体として正しいとは限らない。

コンテクスト

パラメーターにインジェクション

- SELECT id, name FROM users WHERE name LIKE ‘somename%’ LIMIT 10 OFFSET 0 ODER BY ASC;

文字リテラル

数値リテラル

配列、JSON、XMLなど複雑なデータ型リテラルもインジェクション対象

コンテクスト

識別子にインジェクション

カラム名

カラム名

- SELECT id, name FROM users WHERE name LIKE ‘somename%’ LIMIT 10 OFFSET 0 ODER BY ASC;

テーブル名

識別子がパラメーターになっているケースも
少なからずある。例：検索対象カラム
特にストアードプロシージャーは要注意。

変数の識別子を出力処理で完全に無害化するには
エスケープするしか方法がない。
（ハッシュ/配列もバリデーションの一種）

コンテクスト

SQL語句にインジェクション

SQL命令が語句

- **SELECT** id, name **FROM** users **WHERE** name **LIKE** ‘somename%’ **LIMIT** 10 **OFFSET** 0 **ODER BY** **ASC**;

ASC/DESCは
脆弱性の原因となる
語句（命令）の代表格

SQL語句がパラメーターになっているケースも
ある。ASC/DESC以外でも稀に見られる。

変数の語句を出力処理で完全に無害化するには
バリデーションするしか方法がない。（ハッシュ/配列もバリデーションの一種）

SQL文にはコンテクストがある
パラメーター・識別子・語句

識別子

コンテクスト毎に適切な出力！
配列、JSON、XMLなど複雑なデータ型
リテラルもインジェクション対象

- SELECT id, name FROM users WHERE name LIKE ‘somename%’ LIMIT 10 OFFSET 0 ODER BY ASC;

パラメーター

語句（命令）

通常、プリペアードクエリ・プレイスホルダ
が対応するのはパラメーターのみ

セキュリティと
必要十分条件

pならばq　（p⇒q）
が成り立つとき，
「pはqであるための十分条件」
「qはpであるための必要条件」
「p⇒qかつq⇒p、必要十分条件」

ソフトウェア動作が正しければ、
出力が正しい
↓
出力が正しければ、ソフトウェア
動作は正しい

後者は明らかに成り立たない

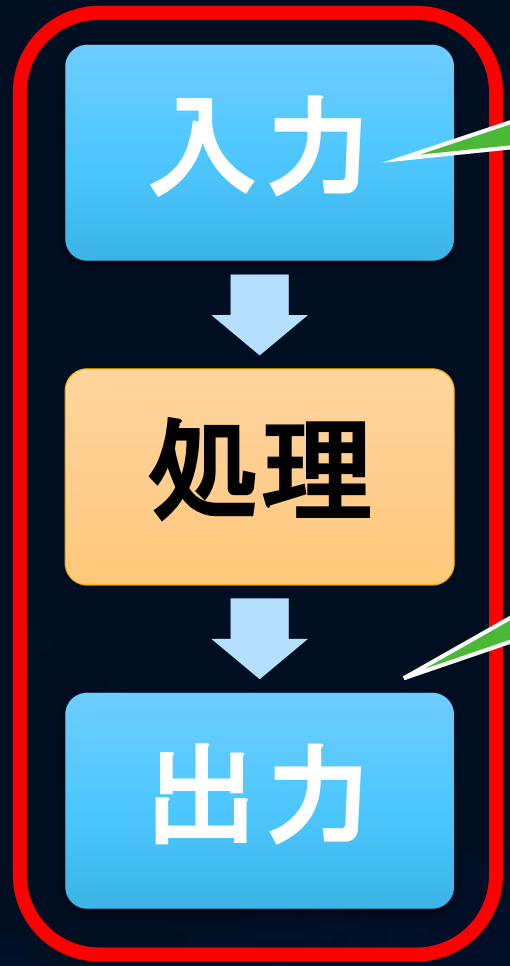
“不正命令インジェクション対策だけ”の
“正しい出力”を求め、
“不正な攻撃文字列”
を含む出力を行うことは
“正しくない”

- SELECT ＊ FROM mytable WHERE id = ’123; DROP TABLE mytable;’
- SELECT ＊ FROM mytable WHERE id = ’123 UNION SELECT * FROM othertable’
- これらは間接インジェクション、論理的なDoSなど別の脆弱性の原因にもなる
- “SQLi”に特化した対策だと、JSONインジェクションなど、複雑なデータ型コンテクストのインジェクションも可能となる

ソフトウェアセキュリティと**必要十分条件**

ソフトウェア動作が正しければ、
入力と出力が正しい
↓
入力と出力が正しければ、
ソフトウェア動作は正しい

- ✓ 入出力が正しければほとんどのインジェクション問題に対応可能
- ✓ 複雑になりすぎるので処理は正しい物とするが、有効であることに変わりない



入力処理では
“入力のコンテクスト“が重要

- ✓ 金額、高さ、幅などの数値
- ✓ 名前、住所、コメントなどの文字列
- ✓ 電話、郵便番号などの定型文字列
- ✓ 都道府県、年代などの分類・集合
- ✓ それぞれのHTTP/SMTPヘッダーなど
- ✓ どのような入力データか？が重要

出力処理では
”出力のコンテクスト”が重要

- ✓ HTML：コンテンツ、URI、JavaScript文字列など
- ✓ SQL：引数、識別子、SQL語句、拡張機能（正規表現、XML、JSON）
- ✓ LDAP、XPath、コマンド、etc
- ✓ どこに出力するデータか？が重要

WAF（Webアプリファイアーウォール）とサニタイズ

- WAFによりSQLインジェクションなどの「インジェクション攻撃リスクは“軽減”可能」
- **サニタイズ**
  ざっくり言うと「不正なデータを綺麗にする」「悪いモノを除去する」ブラックリスト対策
- **セキュリティ対策として“サニタイズ”は基本的に行うべきではない！**
  - **バリデーションと異なり複雑性を増す**
  - **ブラックリスト対策とサニタイズの相性は最悪**

- Oracleの例：
  - “¥0”（ヌル文字）をサニタイズ → “¥0”（ヌル文字）を除去
  - “S¥0EL¥0C¥0T * F¥0ROM some” → “SELECT * FROM some”
  - “¥0”サニタイズがWAFによる防御回避に悪用される
- このような事例は多数ある。
  例）Webブラウザ
  - そもそも**ブラックリスト型対策の信頼性は低い**が、**サニタイズにより更に低くなる**

セキュリティ対策では「ホワイトリスト型対策」が常に優先される理由

なぜ「ブラックリスト」は脆弱なのか？

- 「人間がリスクを正しく判断できない一つは過信であり、もう一つは将来の**あらゆる可能性を徹底して検証する能力の欠如**だ」
- 「**完璧なブラックリスト**」の作成には「**あらゆる可能性を徹底してリストアップする能力**」が必要

サニタイズのように、完璧にしても問題が発生するモノもある！
あらゆる可能性の検証は困難

- 「**完璧なホワイトリスト**」の作成は「**仕様に従った定義を作る能力**」があればOK

「仕様に従った定義」は検証が容易

SQLインジェクションの脅威

高度な攻撃ツールにより、脆弱性は容易に“高度”な攻撃に晒される

今時のSQLインジェクション

一か所脆弱な部分を見つければ、ほぼ自動で

- **全てのテーブルのデータを盗まれる・改ざんされる**
- **任意ファイルをアップロード・ダウンロードされる**
- **任意コマンドを実行される**

今時のSQLインジェクション

- http://sqlmap.org/
- **自動でSQLインジェクション脆弱性に対して高度な攻撃を実行**
  - しかも可能な限り痕跡を残さない
- **ブラインドSQLインジェクション**
  - タイミング攻撃などを利用し、自動でデータベース構造を解析
  - テーブル名やカラム名にプレフィックスを付ける対策は無意味
- **データベースのデータ取得・改ざん**
- **任意ファイルのアップロード・ダウンロード**
- **任意コマンド実行**

どこまで攻撃できるか？は条件（環境）による

ブラインドSQLインジェクション

- タイミング攻撃などを利用
- **サイドチャネル攻撃**の一種
  - 秘密の情報などを保護する暗号化などの仕組みを直接攻撃せず、振る舞いの違い（タイミング/実行時間の差、大きさ、ノイズなど）を利用して攻撃する手法
  - タイミング攻撃以外では圧縮を利用したHTTPS攻撃などが有名
- **タイミング攻撃**はテーブル・カラム名を一文字ずつ解析
- 論理値を使ったブラインドSQLも
  - https://www.owasp.org/index.php/Blind_SQL_Injection
- ブラインドSQLインジェクションに脆弱な場合、
  秘密のDB構造は役立たない

脆弱なパラメーターの種類によるが、入力バリデーションが効果的な場合も多い

インジェクション対策

SQLインジェクションに限らず、“インジェクション対策”は共通

正しいSQLクエリ の必要十分条件

pならばq　（p⇒q）
が成り立つとき，
「pはqであるための十分条件」
「qはpであるための必要条件」
「p⇒qかつq⇒p、必要十分条件」

- **正しいSQLクエリ → データが安全に出力されている**
  - 「データが安全に出力（SQL命令をインジェクションされないように出力）されている」は「正しいSQLクエリ」必要条件にならない
  - 反証：‘ OR ’a‘=’a‘ が電話番号カラム保存されたり、数値検索クエリ条件に利用された場合、正しくないSQLクエリ
  - 「プリペアードクエリを使えばよい」は明らかな間違い

- **正しいSQLクエリ → データは入力コンテクストに対し妥当かつ出力コンテクストに対し安全に出力されている**
  - プログラム中の“処理”が正しいと仮定する場合、**必要十分条件**となる
  - **同じ論理が全てのインジェクション対策に適用できる**

CERT Top 10 Secure Coding Practices
＃１ 入力をバリデーションする

入力バリデーションは“入力コンテクスト”に対して行う

- **全て**の信頼できないデータソースからの**入力をバリデーションする**。適切な入力バリデーションは非常に多くのソフトウェア脆弱性を排除できる。**ほぼ全ての外部データソースに用心が必要**である。これらにはコマンドライン引数、ネットワークインターフェース、環境変数やユーザーが制御可能なファイルなどが含まれる。

正しく動作するための“必要条件”

必要条件を満たさない対策は「不十分」または「非効率」な対策になる

入力対策の2原則

1. ゼロトラスト
2. ホワイトリスト

全ての入力を信頼しない
自分のプログラムからの出力も信用しない
外部からの入力は改ざん可能
安易に「外部入力でない」と考えない

ブラックリスト型はNG

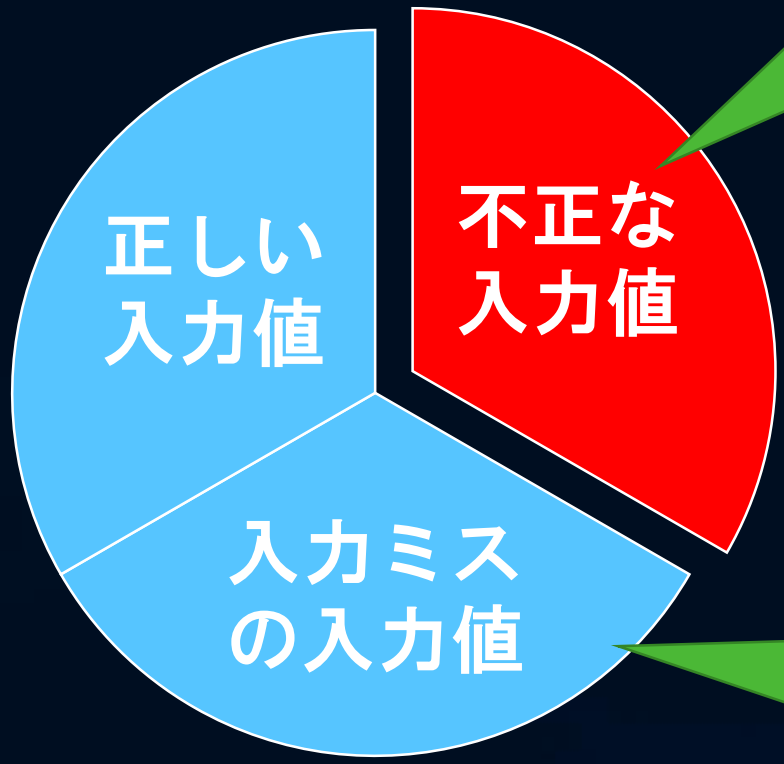
文字エンコーディング、長さ、範囲、文字種、形式、パラメーターの数などを厳格なホワイトリストでバリデーション
ブラックリスト型バリデーションは例外

- 他のインジェクション対策も同じ
- 自分が完全に制御可能なデータ（ハードコードされたデータなど）しか信頼できない
  - ただし、同一プロセス/スレッド上に限る

入力値の種類

入力値には右の三種類しかない

入力バリデーションエラーと
入力ミスエラーの違いは明白
だが、区別ができないケース
も時々見かけるので注意！



入力バリデーション
エラーの入力値は
「拒否すべき入力」
仕様上、あり得ない入力

入力ミスの入力値は
「受け入れるべき入力」
仕様上、あり得る入力

CERT Top 10 Secure Coding Practices
＃7 他のシステムに送信するデータを無害化する

- コマンドシェル、リレーショナルデータベースや商用製品コンポーネントなどの**複雑なシステムへの渡すデータは全て無害化**する。**攻撃者はこれらのコンポーネントに対してSQL、コマンドやその他のインジェクション攻撃を用い、本来利用してない機能を実行できることがある**。これらは入力バリデーションの問題であるとは限らない。これは複雑なシステム機能の呼び出しがどのコンテクストで呼び出されたか入力バリデーションでは判別できないからである。これらの複雑なシステムを**呼び出す側は出力コンテクストを判別できるので、データの無害化はサブシステムを呼び出す前の処理が責任を持つ**。

正しく動作するための“必要条件”

入力対策と出力対策は
独立した対策！
よく間違えられるので
注意！！

出力対策の３原則

出力でもゼロトラストが重要
#0 の対策はゼロトラスト

1. エスケープ
2. エスケープが不必要なAPI
3. バリデーション

エスケープが必要ない（と思われる）APIが、本当にエスケープ無しで安全か知るには、エスケープの知識が必要。セキュリティ対策の文脈ではエスケープ優先。

確認後にAPIの利用を優先することはあり得る。一般にコーディング規約ではエスケープが不必要なAPIを優先することが推奨される。

- 他のインジェクション対策も同じ
- 前提条件

文字エンコーディング、長さ、範囲、文字種、形式などがホワイトリストでバリデーション済み
「文字エンコーディングが正しい」＝「バリデーションされている」＆「設定が正しい」
出力対策と入力対策は独立した対策。入力対策の有無に関わらず出力で完全に無害化

出力対策３の原則と セキュアプログラミング

- セキュアプログラミングと言えばCERT
  - 出力対策は7番目！
- **CERT Top 10 Secure Coding Practices**
  - **＃1 入力をバリデーションする**
  - **＃7 他のシステムに送信するデータを無害化する**

入力対策と出力対策は独立した対策！
よくある間違い
・入力対策はインジェクションを完全に防げないので必須でない
・入力対策しているので、出力対策は省略可能

・文字エンコーディング
・文字種/形式
・長さ/範囲
・パラメータ数など

・エスケープ
・エスケープが不必要なAPI
・バリデーション

正しい実行の“必要十分条件”が揃って正しい実行状態が保証可能

よくある間違い

- 「**入力バリデーションだけでは**
  インジェクションを防げないので
  セキュリティ対策的な意味はない・低い」
  手近な情報による誤謬！

大多数のインジェクション攻撃は“コンテクストに対して妥当ではないデータ”で攻撃する！！
つまり、入力バリデーションにより大多数の攻撃を防止可能！！

- **入力バリデーション**は「入力データがその**コンテクストに対して妥当であることを検証する機能**」
  - 例）「問い合わせ」データの妥当性は、文字エンコーディング、制御文字などの不使用、長さ、で検証される

そもそも不正な入力を受け入れることが間違い

- **出力の無害化**は「出力データがその**コンテクストに対して無害であることを保証する機能**」
  - 例）「問い合わせ」データの出力安全性は出力先（コンテクスト）による。HTMLなら“<“が問題に、SQL文字列引数なら ‘（シングルクオート）が問題になる

不正な入力を受け付けること が根本的な間違い

- SELECT * FROM mytable LIMIT **99999999999**;
- INSERT INTO mytable (col) VALUES ('**<script>alert(''You are hacked :p''</script>**');

SQLiteでは特に高リスク！
カラムのデータ型に関係なく文字列を保存可能

- 入力バリデーションがない出力対策だけでは明らかに不充分
- **「不正な入力」を無視・放置することは脆弱性**
  - 2017年版OWASP TOP 10では第7番目の脆弱性として扱われる

“狭義”の SQLインジェクション対策

不正なSQL文を実行するSQLインジェクション

“狭義”のSQLインジェクション対策

- 「**不正なSQL文の実行対策**」
  - **対策は簡単、「出力対策の３原則」を実施するだけ。**
  - **コンテクストが重要！**
- SQL文の**３つのコンテクスト**
  - **パラメーター（データ）、識別子、語句**
- **APIやストアードプロシージャー**のパラメーター
  - これらの**パラメーターは「データ」「識別子」「語句」の可能性**がある
  - 自分が作る場合、 「出力対策の３原則」を実施するだけ。
  - 他人が作ったモノの場合、シグニチャを見ただけではコンテクストや安全対策の有無は判らない
  - 不適切な利用方法、実装で“狭義”のSQLインジェクションが可能となる場合も多い

「インジェクションによるSQL文実行を防ぐ」“だけ”ならこれだけでもほぼ十分

しかし、「SQL文実行を防ぐ」“だけ”でOKか？！

“狭義”のSQLインジェクション対策

- **前提条件** - 正しい文字エンコーディング
- **パラメーター**
  - SQL標準の文字リテラルは*‘文字列’*、エスケープは‘’
  - パラメーターと命令/識別子を分離できればインジェクション防止
    プリペアードクエリ/プレイスホルダ
- **識別子**
  - SQL標準の識別子クオートは*”識別子”*、エスケープは””
  - **識別子を分離するAPIはない**
- **語句**
  - **バリデーションのみ**
  - 最も多いミスは**ORDER BYのASCとDESC**

コンテクストが重要

標準方式でない物も！
例：MySQLは¥’をサポート

出力対策３原則の＃１がエスケープである理由。エスケープを知らないとAPIが正しいか判らない

標準方式でない物も！
例：MySQLはデフォルトでは`（バッククオート）を利用

少し変わったSQLインジェクション 文字エンコーディングを使った攻撃

- 文字エンコーディングが正しくないとインジェクションに脆弱になる
  - **「データの文字エンコーディング」と「文字エンコーディング設定」が正しくなければならない**
- 文字エンコーディング（UTF-8、SJISなど）処理の実装により
  「壊れた文字エンコーディング」などで誤作動する
- 例：SQL文字リテラル
  ’（マルチバイト文字の最初方のバイト）’
  ’（マルチバイト文字の最初方のバイト）”
  ’（マルチバイト文字の最初方のバイト）¥’
  ’（マルチバイト文字の最初方のバイト）¥¥’
  は正しく処理されるか？
- SJIS系のエンコーディングは危険性が高い
  - **¥でエスケープする処理系はどうすることもできないケースがある**
- 状態を持つエンコーディング（ISO-2022、UTF-7など）は危険性が高い
  - 幸いDBMSでこれらの文字エンコーディングはサポートされていないことが多い
  - 危険なので現在のWebブラウザはサポートしていない

これらが「正しく処理できない」場合に、SQLインジェクションが可能となる
↓
文字エンコーディングのバリデーションが重要

SQL以外でも同じ

追加のインジェクション対策

- **最小権限の原則**
  - DBユーザーの切り替え
    大半のアプリは「全てのテーブルに読み書き」できる
    「スーパーユーザー」のようなアカウントでDBを利用
  - VIEWの利用
- **ホワイトリストによる入力バリデーション**
  - **SQLインジェクションに限らず**
    「ついうっかり」「知らなかった」に対して絶大な威力
- **適切な防御策の実装・実行**
  - **2017年版 OWASP TOP 10 のA7**

ストアードプロシージャーを作れる場合、容易に任意コード実行が可能となる場合も！

セキュアコーディング＃１の対策

A7を要約すると
「入力バリデーションしておかしな不正な入力を検出し、これらのユーザー/システムにアクセスを許すな」

SQLシステムのコンテクストは **SQLのみではない**

- JSONインジェクション
- XMLインジェクション
- XPATHインジェクション
- Regexインジェクション
- LIKEクエリインジェクション
- コマンドインジェクション
- ストアードプロシージャー

SQLデータベースにはこれらのインジェクション攻撃が可能

不正操作のインジェクション以外にDoSに対する防御も必要

物量的DoS
論理的DoS
ReDoSなど

“広義”の SQLインジェクション対策

不正な命令や値のインジェクション対策

“広義”のSQLインジェクション対策

- 「**不正な命令やデータの防止対策**」
  - 狭義は「不正なSQL文の実行防止対策」
- **不正なパラメーター/データ自体のリスク**
  - SELECT * FROM mytable LIMIT **9999999999999999**;
  - INSERT myusers (id, name) VALUES (**0**, ‘attacker’);
- **SQLの「パラメーター/データ」コンテクストは、更に細分化されている**
  - 数値、文字列、JSON、XML、正規表現、部分一致検索（LIKE）など
  - **SQL DBの「パラメーター」を細工すると攻撃が可能になる**

インジェクションされて困るのは「SQL命令」だけではない！

JSONインジェクション

- 現在はJSONデータでJavaScript実行を行う攻撃は困難
- しかし、**JSONに不正なデータを挿入することは可能**
- RailsのMass Assignment（古いが、PHPのregister_globals）と同じ問題が生まれる
  - 例：不必要な追加の値のインジェクションによる認証回避（認証済みフラグのインジェクションなど）
- **対策：JSON文字列を生成する場合、文字リテラルには JavaScript文字列エスケープ、数値リテラルにはバリデーションが必要。JSON文字列の元となる配列、オブジェクトへのインジェクションを許さない**

文字列として生成している場合、SQLコマンドのインジェクションも可能

XMLインジェクション

- HTMLインジェクションとほぼ同じ影響がある
- それに加えて、XXE、プロセスインストラクションやXSLコマンドのインジェクションなどを考慮する必要がある
- XMLもDoS攻撃に脆弱（XDoSなど）
  - https://en.wikipedia.org/wiki/XML_denial-of-service_attack
  - http://www.ws-attacks.org/XML_External_Entity_DOS
- **対策：XMLエスケープ、バリデーション。XML文字列の元となる配列、オブジェクトへのインジェクションを許さない**

文字列として生成している場合、SQLコマンドのインジェクションも可能

XPathインジェクション

- XPathクエリはSQLと同じく、ブラインドXPathインジェクションが可能
  - **つまりXML構造を知らなくても攻撃可能**
  - **全てのデータを簡単に取得される**

一時期「エスケープがインジェクションの原因で悪である」という、とんでもなく誤った認識が蔓延る時期があった

- **対策：エスケープ、バリデーション**
  - XPath 1.0は文字リテラルのエスケープが未定義の欠陥標準
  - XPath 2.0からはSQL標準と同じ方式でエスケープが定義されている

エスケープ書式がある（必要な）のに、エスケープAPIがないインターフェースは欠陥インターフェース

結構多くのDBMS、正規表現など

Regexインジェクション

- 正規表現検索により特定のレコードのみを返すことを想定した正規表現に対し、追加の正規表現で開発者が意図しない検索結果を返す攻撃
- 攻撃の構造・仕組みはSELECTクエリのインジェクションと同等
- 詳しくは
  - https://blog.ohgaki.net/regular-expression-injection
- **Regexと同じくLIKEクエリへのインジェクションが問題となる場合もある**
  - **LIKEクエリのエスケープ漏れは非常に多い間違い**
- **対策：メタ文字エスケープ、バリデーション**

正規表現のエスケープ関数は定義されていない事が多い。LIKEのエスケープ文字は定義可能。

コマンドインジェクション

- 高度な例（共有ライブラリをアップロードしコマンド実行する）以外ではMS SQL Serverのxp_cmdshellが有名
- コマンドを実行する場合、他のプログラミング言語と同様の注意が必要

- **対策：バリデーション、エスケープ**

複数のシェルが選べる環境で、完全に正しくエスケープするのは困難。基本的にバリデーションが必要。コマンドと引数を分離するAPIを利用する場合も、SQLの「物量的DoS」と同様、「値その物」に起因する問題の可能性がある。

ストアードプロシージャー

- ストアードプロシージャーなどのパラメーターが
- **どのようなコンテクストで利用されるか？は実装次第**
- **安全な形で利用されるか？も実装次第**
- **パラメーターが識別子かつエスケープなしでクエリに利用、はよくあるパターン。他のコンテクスト（JSON,XML,正規表現など）に利用されることも普通**

- **対策：仕様（実装）をよく理解した上で安全に利用する。エスケープ、バリデーション**

物量的DoS

- LIMIT句のパラメータが大き過ぎる
- SELECT * FROM table LIMIT 9999999999;
- **対策：バリデーション**
- **数値以外に単純に大きすぎる文字列データも**

単純だがテーブル内のレコード数が多い場合は非常に効果的

データ型のバリデーションは“弱い”バリデーション！

- 単純にリスエスト数が多い
- ab –c 1000 –n 999999999999 http://example.com/
- **対策：リクエスト数の制限・制御**

アプリケーションによる対策は限定的。ネットワークファイアーウォールが効果的。

論理的DoS

- エラーなどを発生させてDoS攻撃を行う
- **論理的な矛盾を起こすデータが原因**
  - 数は多くないが、実際に問題となると厄介
- 例：NUMERIC型で保存したデータをINT8型のカラムに保存
  - DBにより動作が異なる。PostgreSQLではオーバーフローするとエラー、MySQLの場合はデフォルトではオーバーフロー値を保存。

例えば、ネストしたメッセージキューの奥深くでエラーになると厄介

- **対策：論理矛盾を起こすデータを保存しない、させない**
  - アプリ＆DBMSレベルで入力バリデーションを行う
  - DBMSレベルのバリデーションにはCHECK制約などが利用可能

データ型の整合性

- **データ型の不整合による論理的DoS**
- DBカラムと言語のデータ型が表現可能な範囲は必ずしも一致しない
  - NUMERIC、整数サイズ、符号付き整数と符号なし整数、倍精度と単精度浮動小数、4倍精度浮動小数、任意精度整数など
- 型の範囲を超えるデータによるエラー
  - PostgreSQLは整数オーバーフローでクエリエラー
  - MySQLは整数オーバーフローした値が保存される（デフォルト動作）

ReDoS

- 正規表現のアルゴリズムを利用したDoS攻撃
- マッチの繰り返しを行う正規表現には、データに対して指数的なCPU時間が必要となる物がある
  - 例：**(a+)+、([a-zA-Z]+)*、(a|aa)+、(a|a?)+**
- 詳しくは
  - https://blog.ohgaki.net/regex-dos-redos
  - https://blog.ohgaki.net/stackexchange-redos-attack
- **対策：エスケープ、バリデーション**
  - 正規表現パラメーターのエスケープ・バリデーション以外にデータのバリデーションも対策になる。不用意に/複雑な正規表現を使わない、も対策。

ReDoS耐性があるPCREの場合でも問題となるケースも！

攻撃者はどうやって脆弱性を発見するか？

攻撃者が脆弱性を発見する方法

- **ソースコードを読む**

- **脆弱性を発見する“ロケーター文字列”をインジェクションする**
  ‘’;!—‘<XSS>=&{()}
  これはJavaScriptインジェクション用のロケーター文字列だが、
  SQLインジェクションに脆弱な個所にも利用できる

- **“ロケーター文字列のインジェクション“は**
  攻撃ツールで自動化可能

攻撃者が脆弱性を発見する方法

2017年版 OWASP TOP 10
A7 不充分な攻撃対策、
で必要とされる対策

- 開発者が取るべき対策
- “ロケーター文字列”等の“不正な入力” を検出＆拒否する
  - 重要：ブラックリスト型ではなく、ホワイトリスト型で検出すること！！
- 具体的にはユーザーをログアウトさせる、IP/IPブロックを拒否する
- “ロケーター文字列”等を挿入したリクエストは、普通の「入力バリデーション」を実装していれば簡単に検出＆拒否可能

普通の入力バリデーションとは？

- ISO 27000/ISMSで求めている入力バリデーション
  - 範囲外の値
  - データフィールド中の無効文字
  - 入力漏れデータ又は不完全なデータ
  - データ量の上限及び下限からの超過
  - 認可されていない又は一貫しないデータ
- 詳しくは
  - https://blog.ohgaki.net/iso27000-and-input-validation

「データのコンテクスト」で考えると解りやすい！

例）年齢、氏名、電話番号、日付、モノの高さ・幅・重さ

まとめ
“SQL“インジェクション対策

まとめ

- 「プリペアードクエリ/プレイスホルダを使っていればOK」「エスケープ/バリデーションは要らない」は**大間違い**
- 「入力バリデーションは脆弱性を隠す対策なので良くない」は**“セキュリティ対策”を理解していない考え方**
- 「文字エンコーディングのバリデーションなどはライブラリなどが処理すべき」は**セキュアコーディング/ソフトウェアセキュリティ/効率的な実装を理解していない考え方**

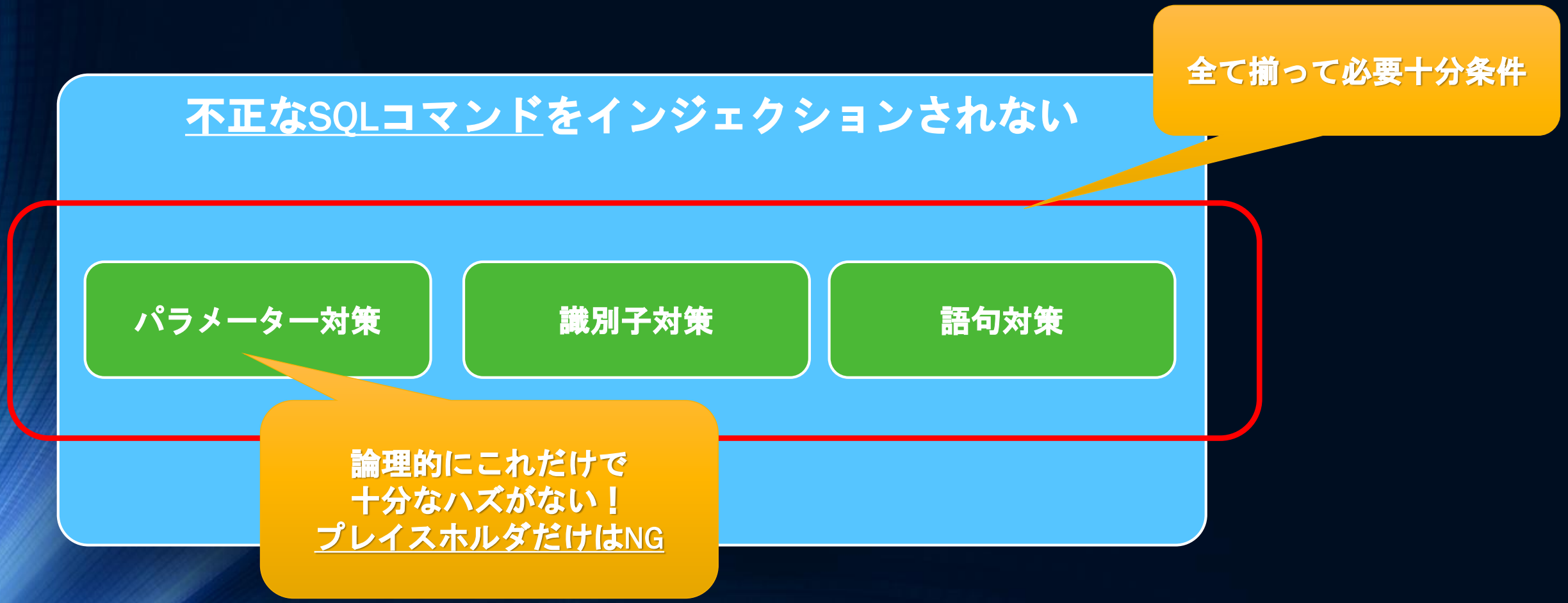
まとめ

- **“コンテクスト”に合わせた出力対策３原則がインジェクション対策**
  - SQLインジェクション以外のインジェクション対策にも**出力対策の３原則**
- **入力と出力、“必要十分条件”を満たす**
  - インジェクション対策では文字エンコーディングのバリデーションは必須項目
- **“入力バリデーション”はソフトウェアが正しく動作する為の前提条件**
  - 前提条件を満たしていないデータに対して、
    正しく、効率的かつセキュアに動作するコードを書くことは困難か不可能
- 一文字（１バイト）でも“意味”がある文字（バイトデータ）がある場合、
  ほぼ確実にインジェクションのリスクがある

まとめ

- “狭義”のSQLインジェクション対策だけでは、
  “広義”のSQLインジェクションは防げない
  - DBMSに保存されたXMLデータをXPathインジェクションで盗まれる
  - JSONインジェクションで不正操作・乗っ取りを許す
  - 様々な手法でDoS攻撃されシステムが停止する、etc
- “狭義”/”広義“SQLインジェクションの区別に意味はない
  - **開発者にとってインジェクション攻撃を“狭義”/”広義“と区別しても、対策は同じ →コンテクストに応じた入力バリデーション/出力対策３原則**
- **「全てのインジェクションを防ぐ方法」を知ることが重要！**

“狭義”のSQLインジェクション対策の必要十分条件



まとめ
“全て”のインジェクション対策

インジェクション対策はユニバーサル
SQLインジェクションは“インジェクションの一つ”に過ぎない

まとめ – 全てのインジェクションを防ぐ

- インジェクションを知る
  - 一文字/１バイトでも、意味がある場合、インジェクションが可能と想定する
  - 意図しない要素が追加できる場合、インジェクションが可能と想定する
  - （このスライドでは解説していませんが）位置に意味がある場合、インジェクションが可能と想定する
- 全ての入力をバリデーション
  - ホワイトリスト型 – データ型のバリデーションは“弱い”バリデーション
  - 文字エンコーディングをバリデーションする
- 出力対策３原則 - 全ての出力を無害化
  - “出力のコンテクスト“に応じた ①エスケープ　②API　③バリデーション

特に“文字列型”であることを保証してもあまり意味がない。
「入力のコンテクスト」でバリデーション！

入力・出力対策、コード（ライブラリなど）は
ゼロトラスト（何も信用しない）が原則

ソフトウェアの構造とセキュリティ

「最低限」でも全ての外部入力・出力の安全対策は必須

これらの内部処理は追加・削除・変更が発生。いつ新たな問題が追加・発見されるかわからない

そもそも入力・処理・出力の実装が「危険」であり、それが仕様のモノも多数存在する

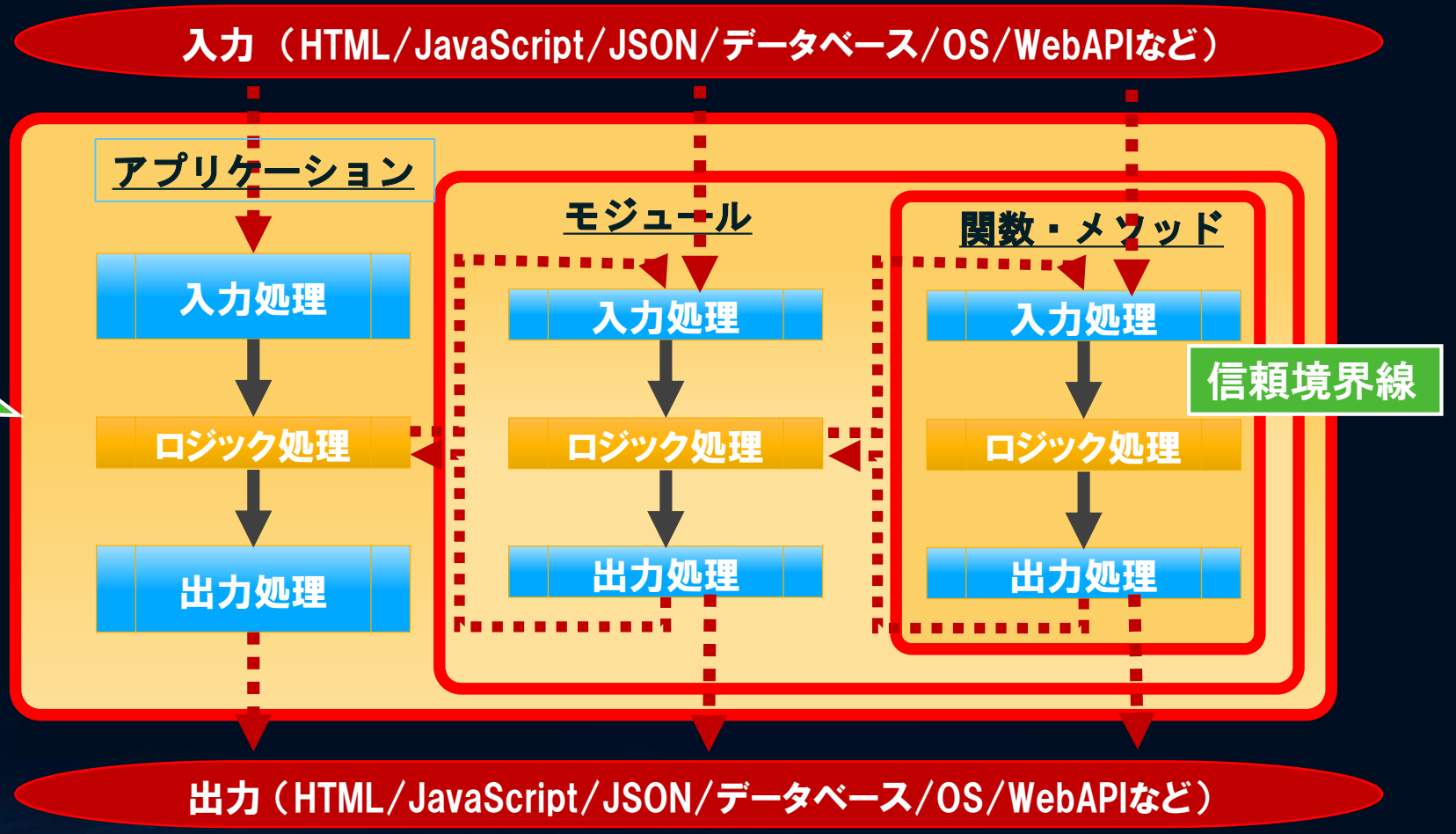
基本的にデータは信頼できない！
未検証のモジュール・APIも信頼できない！

ゼロトラスト！

データはバリデーション後も信頼できない！

セキュアなソフトウェアアーキテクチャ

- 基本アーキテクチャ　– アプリケーション、モジュール、関数・メソッド、粒度に関わらず共通



各レイヤーの
境界防御
が防御の基本

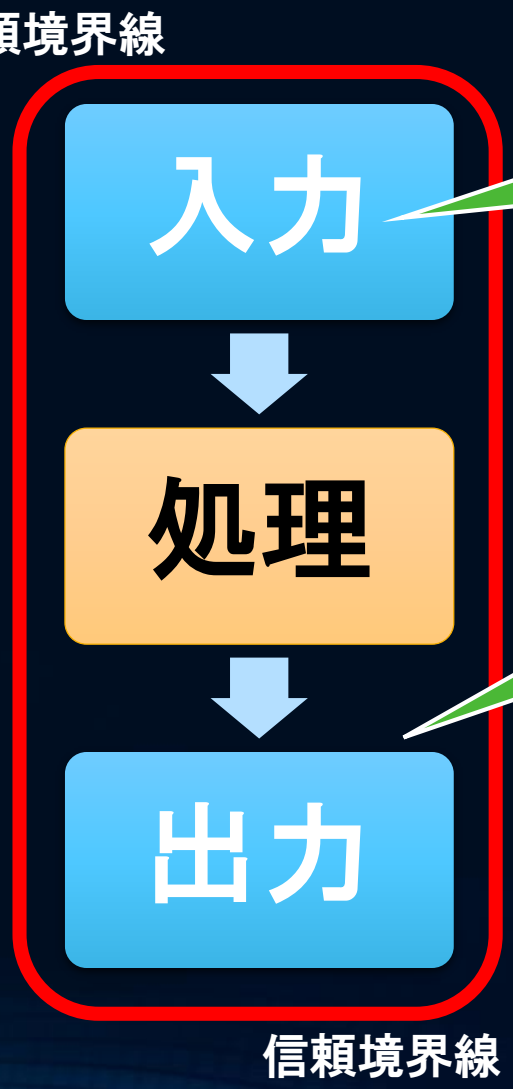
- ✓ アプリケーションレベルの境界防御が特に重要
- ✓ データは基本的に信頼できない
- ✓ アプリレベルで入力バリデーションすれば、アプリ全体に対して妥当性を保証可能

セキュリティ対策と<u>コンテクスト</u>

どちらもコンテクストが重要！

- **入力処理**
  - <u>入力のコンテクスト</u>で バリデーション
- ✓ アプリケーションの入力バリデーション（ホワイトリスト型）が最も重要
- ✓ 重要なモジュール/関数は縦深（多層）防御

- **出力処理**
  - <u>出力のコンテクスト</u>で、エスケープ、安全なAPI、バリデーション（ホワイトリスト）
- ✓ できる限り“最終出力”に近い場所でセキュリティ処理を行う
- ✓ “最終出力”に近い場所の方が“コンテクスト”が明確



入力処理では “<u>入力のコンテクスト</u>”が重要

- ✓ 金額、高さ、幅などの数値
- ✓ 名前、住所、コメントなどの文字列
- ✓ 電話、郵便番号などの定型文字列
- ✓ 都道府県、年代などの分類・集合
- ✓ それぞれのHTTP/SMTPヘッダーなど
- ✓ **<u>どのような入力データか？が重要</u>**

出力処理では “<u>出力のコンテクスト</u>”が重要

- ✓ HTML：コンテンツ、URI、JavaScript文字列など
- ✓ SQL：引数、識別子、SQL語句、拡張機能（正規表現、XML、JSON）
- ✓ LDAP、XPath、コマンド、etc
- ✓ **<u>何処に出力するデータか？が重要</u>**

信頼境界線とコンテクスト

- 信頼境界線でもコンテクストが重要！

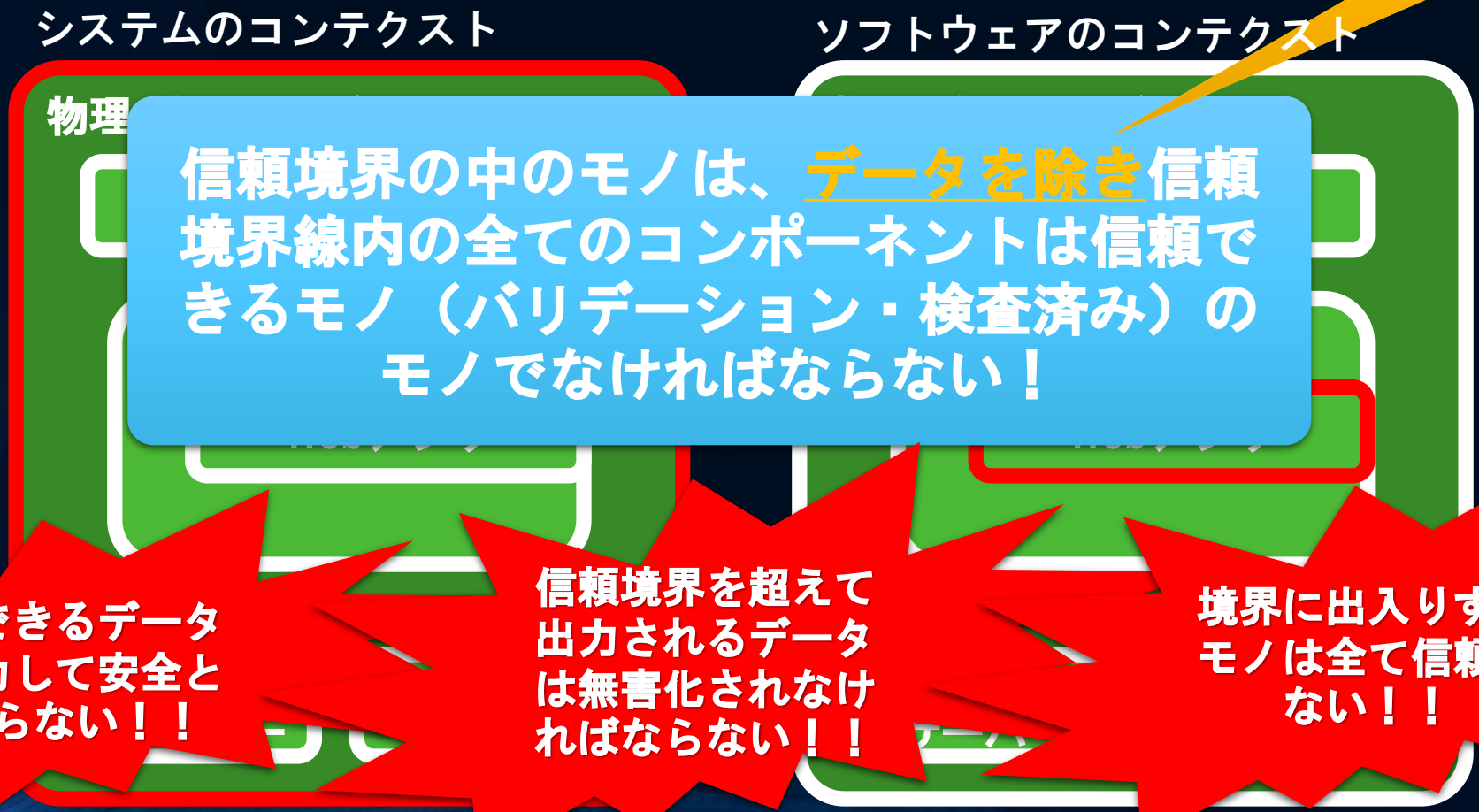
大雑把な
イメージ！

基本となる信頼境界線のイメージ

信頼境界線とコンテクスト

- 信頼境界線でもコンテクストが重要！

基本となる信頼境界線のイメージ

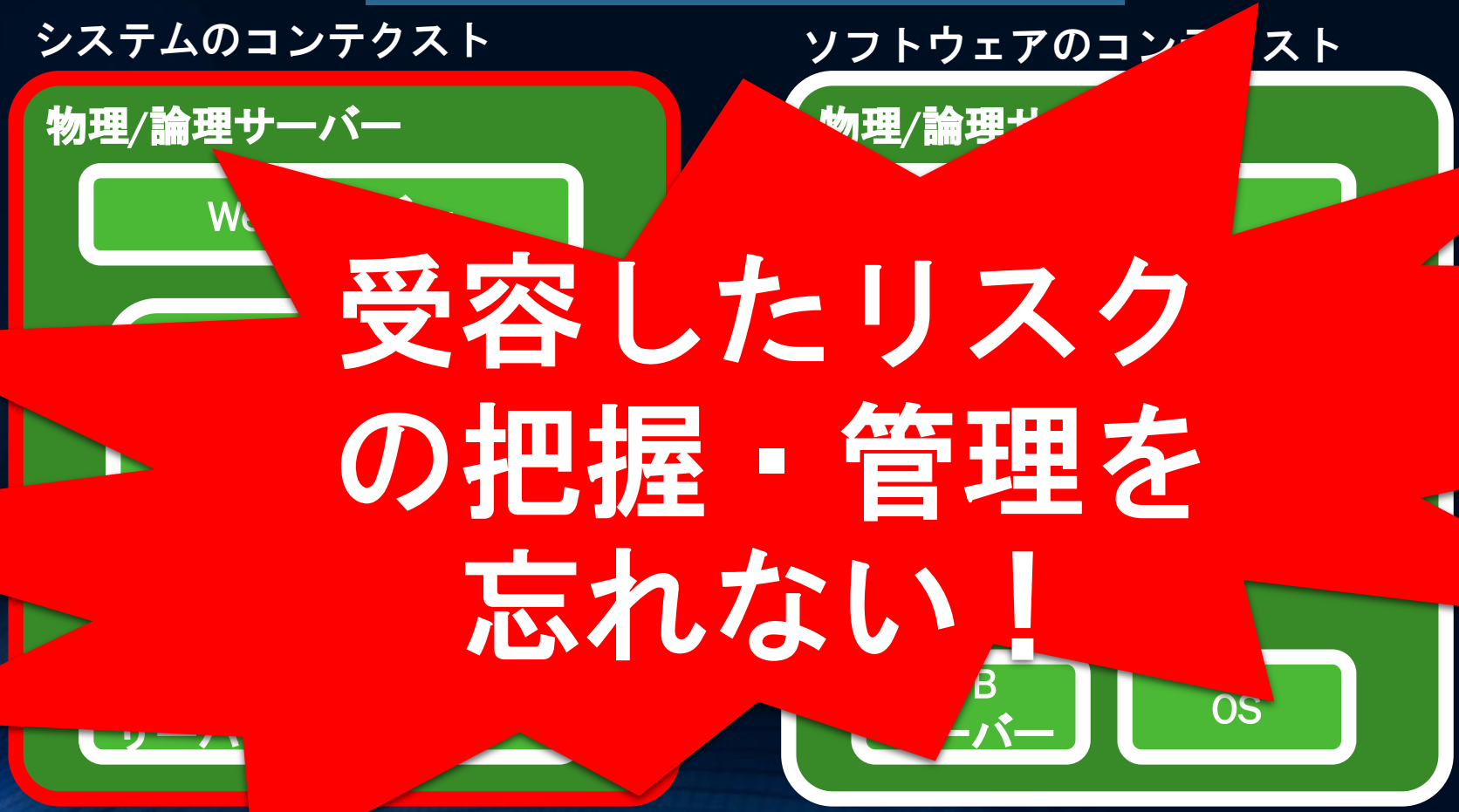


信頼境界線とコンテクスト

大雑把な
イメージ！

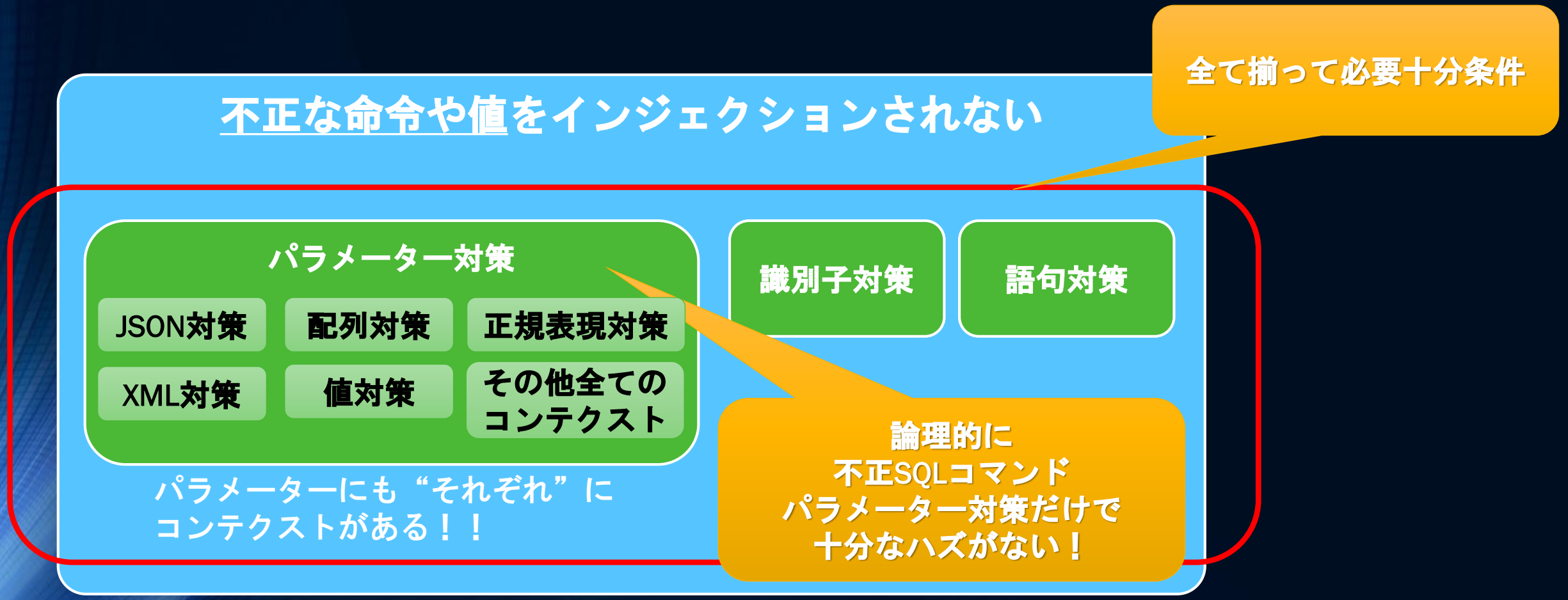
- 信頼境界線でもコンテクストが重要！

基本となる信頼境界線のイメージ



受容したリスク
の把握・管理を
忘れない！

“広義”のSQLインジェクション対策の必要十分条件



正しい実行結果となるコードとは？
＝セキュアなコードとは？

- 50年代　LISP
- 80年代　契約プログラミング（契約による設計）
- 90年代　防御的プログラミング
- 2000年代　セキュアコーディング
- 1988年のモリスワーム以降、入力と出力の正しさがソフトウェアセキュリティ的にも重要と認識される
- チューリングマシンで有名なアラン・チューリングを入れると第二次世界大戦より前から研究されている　例）停止性問題

セキュアなコードの必要十分条件（入力出力対策）は皆さんが生まれる前から！
コンピューターサイエンティストはそもそも数学者だったので“必要十分条件”は常識中の常識

入力バリデーションの注意事項

- バリデーションの基本は**ホワイトリスト**
- **データ型の保証は“弱い”バリデーション**
- このスライドいう
  「入力ミスによるエラー」と
  「入力バリデーションエラー」は
  全く別のモノです！
- 入力バリデーションを行う“**前**”に
  正規化などの変換処理を行う
- 入力バリデーションをなった“**後**”に
  正規化などの変換処理を**行わない**

データ型保証に“不可能な期待”をしすぎ。データ型保証では正しいプログラムの実行を保証できない！

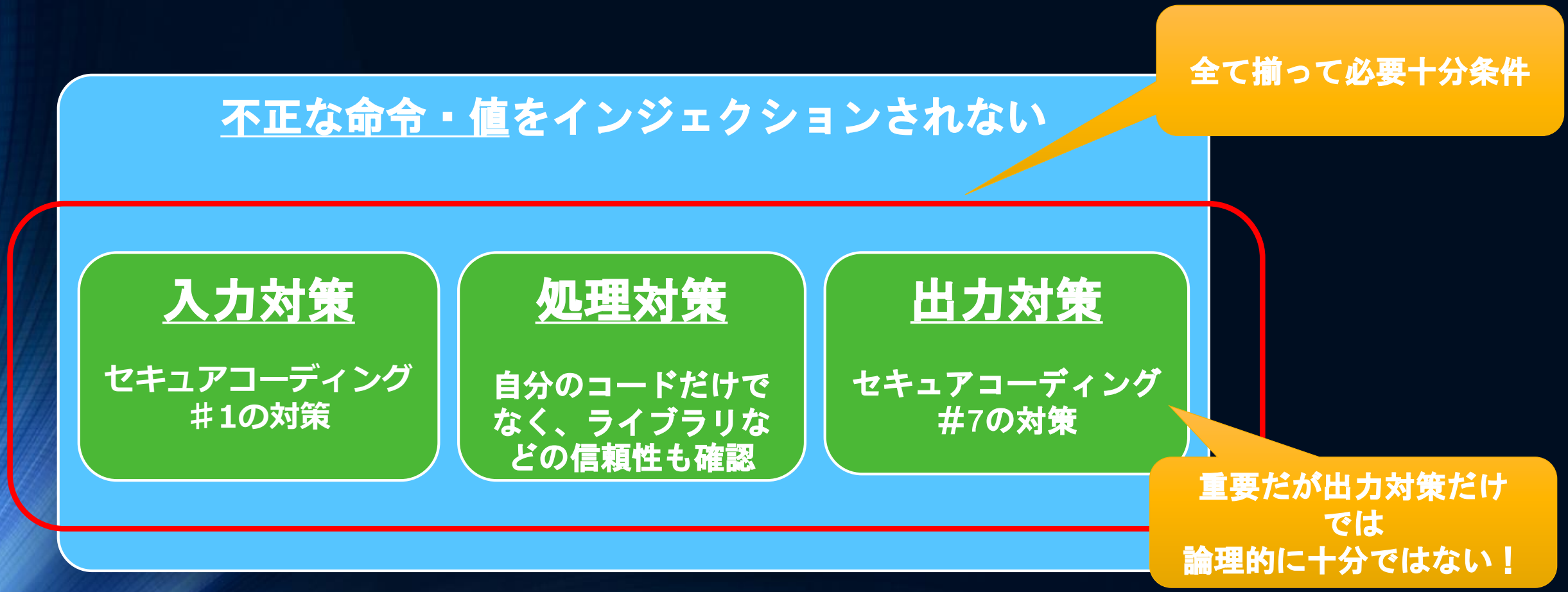
よくある間違いなので注意！
「入力ミスによるエラー」は
仕様上*あり得る*エラー
「入力バリデーションエラー」は
仕様上*あり得ない*エラー

変換処理と入力バリデーション処理の順序を間違えると脆弱になる！

論理的なセキュリティ対策！

- **論理的に破綻しているセキュリティ対策が「難しい」のは当たり前**
  - 「場当たり的対策」「何をすればよいのか解らない」「どこまですれば良いのか解らない」「論理的に整合性が取れないので解らない」
- 初心者なら「How Toレベル」でもOK！
  - 「プリペアードクエリを使う」「安全なAPIを使う」
- 中級者以上は「**論理的なセキュリティアーキテクチャー**」を考える！
  - セキュアなアーキテクチャーのコードはセキュリティの維持管理も、コードの変更も容易

セキュアなアプリケーションの必要十分条件



結論：**セキュアコーディング**しましょう！
基本に忠実な方が**楽**で**効果的**！

- CERT TOP 10 Secure Coding Practices
  - https://blog.ohgaki.net/cert-top-10-secure-coding-standard
- ISO 27000/ISMSの入力バリデーション
  - https://blog.ohgaki.net/iot-security-1st-is-validation
- セキュアプログラミングの７つの習慣
  - https://blog.ohgaki.net/the-7-habits-of-secure-programming
  - 「**ゼロトラスト**」は**全てのセキュリティ対策の基本中の基本**
- 契約による設計と信頼境界線
  - https://blog.ohgaki.net/design-by-contract-and-trust-boundary

残念ながら、現在は何十年も前から提唱されている
「セキュリティの基礎の基礎」
「入力バリデーション」
さえできていない状態！

“ゼロベース”とは言っても、一から自分で考える必要はない。
セキュリティの基礎の基礎から最適なセキュリティを再構築する。

論理・基礎概念に則ったセキュリティ対策が必要！

ソフトウェアセキュリティは一度ゼロベース、考え方をリセットして考え直した方が良い

過去からの不合理な習慣、不合理な概念、不充分・不安定・不合理な対策、論理的に正しいセキュリティの方が簡単かつ解りやすく効率が良い。残念ながら“専門家”と言われる人でも非論理的な対策を“対策”としている場合も多い。

ご清聴ありがとうございました。

弊社ではソースコード検査/研修/テクニカルサポートなどの
サービスも提供しています。