チートを未然に防ぐ脆弱性診断

業務部
柴田 有

概要

- 脆弱性診断とは？
- 脆弱性診断の進め方
  - ガチャのサンプルコードから脆弱性を探してみよう
- まとめ
  - 脆弱性の原因は？
  - 脆弱性を減らすために開発時にできることは？

脆弱性診断とは？

- ゲームに脆弱性がないか診断する
  - 脆弱性とは？
    - 情報漏洩
      - 認証情報・個人情報など
    - アプリクラッシュ
    - ゲームチート
      - 無敵・アイテムの増殖など
    - などなど、想定外の挙動全般
  - このスライドでは「脆弱性 ＝ ゲームチート」

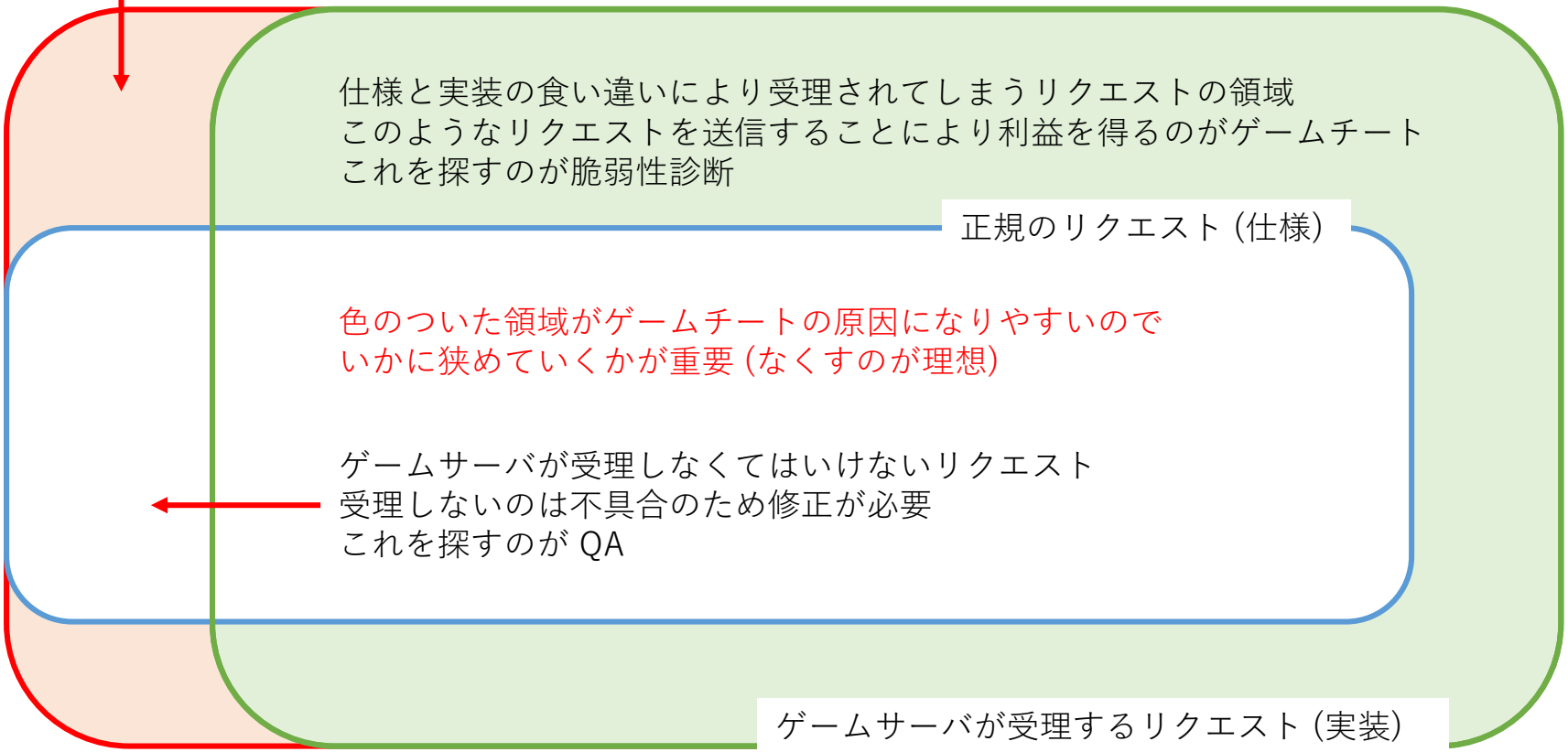
診断のスコープ

- クライアントアプリ
  - 攻撃者の手元にあるので、理論上はなんでもできる
- サーバアプリ
  - 直接操作できない
    - 不正なリクエストを送信して誤動作させる
      - どのような？
      - そもそも不正とは？
    - ゲームサーバに侵入可能か？

このスライドのメインスコープ

不正なリクエストのイメージ

不具合を修正する際にゲームサーバが受理する範囲を必要以上に拡大してしまったケース
不具合だけでなくゲームチートについても配慮しないと、ゲームチートの原因になりやすい
不具合の修正時期が前後してしまうと、脆弱性診断の対象外になってしまうことも



ゲームチートを防ぐには？

- 不正なリクエストを受理しないようにする
  - 仕様と実装の違い (色のついた領域) をなくす
  - これを確認するのが脆弱性診断の目的
- 不正なリクエストを送信しにくくする
  - 通信データの暗号化や改ざん検知などにより、色のついた領域をさわりにくくする
  - クライアントアプリの解析などが必要になるため、ゲームチートの敷居を高くする効果がある

脆弱性診断の作業内容

- 色のついた領域の中から脆弱性を探す
  - 何をするとどのような利益が得られるか？
- 脆弱性の修正作業
  - 開発者に脆弱性の内容を指摘して修正してもらう
    - 実装を仕様に寄せていく
  - 修正により問題が解消されているか確認する

脆弱性診断の手法

- ブラックボックス診断
  - 既知の攻撃リクエストを機械的に試して、問題が起きなければ実装が正しいと判断する
    - 過去のゲームチートの知見の積み重ねが重要
- ホワイトボックス診断
  - 仕様と実装を確認して、問題が発生しそうなリクエストを中心に検証する
    - 実装の確認のためにソースコードを読む
    - 実際にリクエストを送信して動作を確認することにより、実装に対する理解を深めていく
      - ログやデータベースの内容なども適宜確認

脆弱性を探してみよう

脆弱性を探す

- ガチャのサンプルコードを提示するので、脆弱性を探してみましょう
  - 脆弱性を探すには？
    - 仕様と実装の違いに着目する
      - 「この仕様ならこう実装するはず」
      - 「どうしてこんな実装になっているのだろう？」
    - 仕様を把握していないと実装がイメージできないので、脆弱性を見逃してしまう可能性がある

今回のガチャの仕様と設計

- ガチャにはいくつか種類があります
  - ガチャごとに消費するリソースは違います
    - アイテムとかポイントとか色々
  - ガチャの種類をパラメータで指定します

- ポイントガチャの仕様
  - ポイントはゲームをしているともらえます
  - それなりにもらえる想定なのでまとめて使えます
    - ガチャの実行回数をパラメータで指定します

脆弱性を探してみよう

```
function playGacha( $gachaId, $playCount ) {

  /* $gachaId 関連の処理 (マスターデータ取得) がこの辺にあると思ってください */

  $player->point -= $unitCost * $playCount;

  for( $counter = 0; $counter < $playCount; ++$counter ) {
    /* ガチャの抽選ロジックと付与処理がこの辺にあると思ってください */
  }

}
```

脆弱性がこの辺にあると思ってください

$playCount が負数の場合

- 消費するポイントが負数になる
  - 所持ポイントから負数減らすとポイントは増える
- ガチャのループ回数が負数になる
  - ガチャは実行されません

- つまり？
  - ガチャの実行回数を負数にするとポイントが増えるので、増えたポイントで好きなだけガチャができる

仕様と実装の違いに注意する

- 負数について言及していない仕様書が悪い？
  - 現実世界で回数が負数になることはないはず
    - そもそもガチャの実行回数が負数とかおかしい
    - そんなことまで書いてある仕様書読みたいですか？
  - 負数が指定できるのは仕様というよりは実装の問題なので、負数が指定できないように実装を見直す
    - 負数を受け付けてしまう実装は、仕様よりも大きなものになっている (色のついた領域ができている)

負数検査を入れて万事解決？

```
function playGacha( $gachaId, $playCount ) {

  /* $gachaId 関連の処理 (マスターデータ取得) がこの辺にあると思ってください */

  if( $playCount < 0 ) {
    throw new Error( “ガチャの実行回数が負数とかおかしい” );
  }

  $player->point -= $unitCost * $playCount;

  for( $counter = 0; $counter < $playCount; ++$counter ) {
    /* ガチャの抽選ロジックと付与処理がこの辺にあると思ってください */
  }

}
```


脆弱性がこの辺にあると思ってください

$playCount が小数の場合

- たとえば $playCount が 0.1 なら？
  - 消費するポイントは単価の 0.1倍になる
  - ガチャのループ回数は切り上げられる
    - ガチャは一回実行される
  - つまり？
    - ガチャを 9割引きで実行できる
    - $playCount をもっと小さくすれば、その分だけ値引きされる (ガチャの価格を自由に安くできる)

仕様と実装の違いに注意する

- 小数について言及していない仕様書が悪い？
  - 現実世界で回数が小数になることはないはず
    - そもそもガチャの実行回数が小数とかおかしい
    - そんなことまで書いてある仕様書読みたいですか？
  - 小数が指定できるのは仕様というよりは実装の問題なので、小数が指定できないように実装を見直す
    - 小数を受け付けてしまう実装は、仕様よりも大きなものになっている (色のついた領域ができている)

型キャスト入れました

```
function playGacha( $gachaId, $playCount ) {

  /* $gachaId 関連の処理 (マスターデータ取得) がこの辺にあると思ってください */

  $playCount = (int)$playCount;
  if( $playCount < 0 ) {
    throw new Error( “ガチャの実行回数が負数とかおかしい” );
  }

  $player->point -= $unitCost * $playCount;

  for( $counter = 0; $counter < $playCount; ++$counter ) {
    /* ガチャの抽選ロジックと付与処理がこの辺にあると思ってください */
  }

}
```

たまにあるダメな修正例

```
function playGacha( $gachaId, $playCount ) {

  /* $gachaId 関連の処理 (マスターデータ取得) がこの辺にあると思ってください */

  $intPlayCount = (int)$playCount; /* 入力パラメータは read only にしたい */
  if( $intPlayCount < 0 ) {
    throw new Error( “ガチャの実行回数が負数とかおかしい” );
  }

  $player->point -= $unitCost * $intPlayCount;

  for( $counter = 0; $counter < $playCount; ++$counter ) {
    /* ガチャの抽選ロジックと付与処理がこの辺にあると思ってください */
  }

}
```

実装を見直す

- 負数や小数が指定できるのはなぜか？
  - 採用したプログラミング言語の仕様
    - プログラミング言語を変更するのは非現実的
  - 指定されたときにエラーになるように検査する
    - それって API の仕事ですか？
      - 全部の API で検査することになると、検査漏れしそう
      - 不具合があったときに修正漏れしそう
      - 全 API で共通の処理をするのがフレームワークでは？

フレームワークでの型検査

- API で使うパラメータの情報を渡しておけば、フレームワークが検査しておいてくれる
  - playGacha API の場合
    - gachaId は整数です
    - playCount は非負の整数です
  - 後は API ロジックの実装に専念できる
    - 考慮することが減ってミス (脆弱性) も減るはず！

フレームワークに任せた結果

```
function playGacha( $gachaId, $playCount ) {

  /* $gachaId 関連の処理 (マスターデータ取得) がこの辺にあると思ってください */

  $player->point -= $unitCost * $playCount;

  for( $counter = 0; $counter < $playCount; ++$counter ) {
    /* ガチャの抽選ロジックと付与処理がこの辺にあると思ってください */
  }

}
```

静的型付けがあると？

フレームワークを使わなくても負数や小数などを気にしなくて良くなる

```
void playGacha( int gachaId, unsigned int playCount ) {

  /* gachaId 関連の処理 (マスターデータ取得) がこの辺にあると思ってください */

  player->point -= unitCost * playCount;
```


脆弱性がこの辺にあると思ってください

```
  for( int counter = 0; counter < playCount; ++counter ) {
    /* ガチャの抽選ロジックと付与処理がこの辺にあると思ってください */
  }

}
```

playCount がとても大きい場合

- たとえば unitCost が 100 のときに、playCount が 42,949,673回なら？
  - 32ビット整数があふれて消費ポイントが 4 になる
  - ガチャは 42,949,673回実行される
    - 多分途中で何らかのエラーが発生します
  - ※静的型付けが問題の原因という話ではありません
    - ゲーム仕様として上限値を正しく設定しましょう

フレームワークの行う検査

- 型検査
  - 静的型付けがあるとこれが楽になる
    - 考慮することが減ってミス（脆弱性）も減るはず！
- パラメータ検査
  - ゲーム仕様に適した有効な値なのか？
    - 仕様と実装を一致させるために重要なメインタスク

ガチャの仕様に追記しました

- ガチャにはいくつか種類があります
  - ガチャごとに消費するリソースは違います
    - アイテムとかポイントとか色々
  - ガチャの種類をパラメータで指定します

- ポイントガチャの仕様
  - ポイントはゲームをしているともらえます
  - それなりにもらえる想定なのでまとめて使えます
    - まとめて実行は 10回までです
    - ガチャの実行回数をパラメータで指定します
  - 一日一回無料でできます

一日一回無料ガチャの追加

```
function playGacha( $gachaId, $playCount ) {

  /* $gachaId 関連の処理 (マスターデータ取得) がこの辺にあると思ってください */

  /* 一日一回無料ガチャを追加しました！ */
  if( $player->isGachaFirstTimeToday( $gachaId ) ) {
    $player->setGachaFirstTimeToday( $gachaId, false );
  }
  else {
    $player->point -= $unitCost * $playCount;
  }

  for( $counter = 0; $counter < $playCount; ++$counter ) {
    /* ガチャの抽選ロジックと付与処理がこの辺にあると思ってください */
  }

}
```

脆弱性を追加しました！
$playCount = 1; が必要

新しい機能を追加するときは

- 既存の機能と衝突しないか考える
  - 一日一回無料 (新規) と回数指定 (既存)
  - 衝突が大きい (複雑な) 場合には機能を分けると、考慮することが減ってミス (脆弱性) も減るはず！

  - 似たような機能なので実装を使い回したい
    - 既存の実装に新機能の実装を継ぎ足すと複雑になるので、よほどミスしない自信がない限りは避けた方が良いです
      - 他の誰かが触る可能性があるなら特に注意が必要
    - 既存の実装を再利用する場合には、コピーするのではなく共通部分をサブルーチンに切り出しましょう
      - コピーしてしまうと修正漏れが発生しやすくなります

ここまでのまとめ

- ガチャの実行回数に起因する問題がいくつか
  - 問題の原因は仕様と実装が違うことで、
    その違いをなくすためにもパラメータ検査は重要
    - フレームワークにパラメータ検査を任せられると、
      API ではメインロジックの実装に注力できるようになる
      - 考慮することが減ってミス (脆弱性) も減るはず！

もう一つのパラメータ

- $gachaId に起因する脆弱性は？
  - 存在しないガチャを指定しても何も起きないので、$playCount に比べると問題は起きにくい
  - 想定していないガチャが実行できてしまう
    - デバッグ用に無料のガチャが登録されていて実行できる
    - チュートリアル用の無料ガチャが何度も実行できる
    - ステップアップガチャの最終ステップだけ実行できる
- 仕様 = クライアントアプリで表示される
- 実装 = リクエストすると実行できる

パラメータ以外の脆弱性

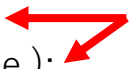
```
function playGacha( $gachaId, $playCount ) {

  /* $gachaId 関連の処理 (マスターデータ取得) がこの辺にあると思ってください */

  /* 一日一回無料ガチャを追加しました！ */
  if( $player->isGachaFirstTimeToday( $gachaId ) ) {
    $player->setGachaFirstTimeToday( $gachaId, false );
    $playCount = 1;
  }
  else {
    $player->point -= $unitCost * $playCount;
  }

  /* ガチャの抽選ロジックと付与処理がこの辺にあると思ってください */

}
```



脆弱性がこの辺にあると思ってください

同時にリクエストされた場合

- たとえば同じプレイヤーから 3個のリクエストがほぼ同時に届いたら？
  - R1「無料ガチャ実行済み？」 - 「まだです」
  - R2「無料ガチャ実行済み？」 - 「まだです」
  - R1「では、無料でガチャします」
  - R3「無料ガチャ実行済み？」 - 「済みました」
  - R2「では、無料でガチャします」
  - R3「では、通常料金でガチャします」
  - この例ではガチャが無料で 2回実行できる
    - もっとたくさん同時にリクエストすれば、無料で実行できる回数も増えるかも？

問題の原因と対応方針

- 問題の原因は？
  - データの確認と更新が分離されていること
    - 実行済みに更新されるまでに確認処理を行えば、何度でも無料でガチャが実行できる
- 対応方針は？
  - 途中で他の処理が行われないように排他制御する
    - リクエストを一つずつ処理すれば、無料ガチャが何度も実行されることはない
      - ただし、他のリクエストが完了するまで待つ必要がある

排他制御のやりかた

- データごとにやる (細粒度)
  - 待ち時間を短くできる
    - 自分が無料ガチャを実行済みか確認すると待たされる
  - 実装は大変
- ある程度まとまった単位でやる (粗粒度)
  - プレイヤーやギルドなどの大枠で考えれば良いので実装はシンプル
    - 考慮することが減ってミス (脆弱性) も減るはず！
  - 待ち時間は長くなる
    - 自分が何かリクエスト中だと待たされる

排他制御は誰がどうやるか？

- キャッシュ (Redis など) の場合
  - 個々のデータはデータベースにあるので粗粒度に
  - 基本的には API のロジックに入る前に実施する
- データベースの場合
  - 粗粒度の場合
    - データベースアクセスを開始する際に実施する
  - 細粒度の場合
    - 個々のデータにアクセスする際に実施する

ダメな例：キャッシュ編

```
function dispatcher( ) {

  if( Cache::hasKey( $playerCacheKey ) ) {
    throw new Error( 'API を実行中です' );
  }
  Cache::insert( $playerCacheKey );

  /* 実行したい API を呼び出す処理がこの辺にあると思ってください */

  Cache::delete( $playerCacheKey );

}
```

同じ失敗の繰り返し
Redis の場合 setnx を使って
一つの命令にまとめる

処理が abort しても消えることを確認しておきましょう

ダメな例：データベース編

```
BEGIN;
-- メンテナンス突破プレイヤーかチェック
SELECT isMaintenanceFree FROM player WHERE playerId = 123;
-- 更新系の API か確認して、フレームワークでプレイヤー単位 (粗粒度) の排他制御
SELECT * FROM player WHERE playerId = 123 FOR UPDATE;
-- API 実行
-- データ単位 (細粒度) での排他制御はしません
SELECT isGachaFirstTimeToday FROM playerGacha
  WHERE playerId = 123 AND gachaId = 456;
-- 以下一日一回無料ガチャ未実行の場合のみ
UPDATE playerGacha SET isGachaFirstTimeToday = true
  WHERE playerId = 123 AND gachaId = 456;
COMMIT;
```

最初のクエリを実行した時点のデータを取得してしまう (REPEATABLE READ) トランザクション開始直後に排他制御を行う必要がある

排他制御の動作確認方法

- 他の処理がどのタイミングで実行されても問題が起きないことを確認したい
  - 同時に複数のリクエストを送信した場合に、どういう順番で処理が行われるか分からない
  - シングルステップ実行できれば全パターン試せるが、検証コストがかなり大きくなる
    - サーバアプリを改変するのも困難
    - 問題が起きそうなタイミングに限定しても良いので、狙ったタイミングで他の処理を実行させるには？
      - 何らかの方法で処理を止めてしまえば良い

Sleep で止める

```
function dispatcher( ) {

  if( Cache::hasKey( $userCacheKey ) ) {
    throw new Error( 'API を実行中です' );
  }
  sleep(1);
  Cache::insert( $userCacheKey );
  sleep(1);

  /* 実行したい API を呼び出す処理がこの辺にあると思ってください */

  Cache::delete( $userCacheKey );

}
```

一秒以内に次のリクエストを送信すればテスト可能
この場合、ダメな例であることが確認できる
ここだと遅いので、テストとしては不十分
排他制御がうまくいっているように見える

Write Lock で止める

```
BEGIN;
-- メンテナンス突破プレイヤーかチェック
SELECT isMaintenanceFree FROM player WHERE playerId = 123;
-- 更新系の API か確認して、フレームワークでプレイヤー単位 (粗粒度) の排他制御
SELECT * FROM player WHERE playerId = 123 FOR UPDATE;
-- API 実行
-- データ単位 (細粒度) での排他制御はしません
SELECT isGachaFirstTimeToday FROM playerGacha
  WHERE playerId = 123 AND gachaId = 456;
-- 以下一日一回無料ガチャ未実行の場合のみ
UPDATE playerGacha SET isGachaFirstTimeToday = true
  WHERE playerId = 123 AND gachaId = 456;
COMMIT;
```

事前にこのレコード Write Lock
複数のリクエストが待っている
状態で Write Lock を解除する

排他制御のまとめ

- 同時にリクエストが送信されると？
  - データが更新されるまでに複数の処理が行われると問題が起きる可能性がある
    - データを更新する際には排他制御をする必要がある
- 排他制御が機能していることを確認するには？
  - 任意のタイミングで処理を停止させて他の処理が行われないことを確認する
  - 粗粒度の排他制御の場合、実装も確認も簡単になる
    - 考慮することが減ってミス（脆弱性）も減るはず！

まとめ

- ゲームチートを未然に防ぐには？
  - 不正なリクエストを受理しないように仕様と実装を一致させる
    - 仕様上正しいパラメータのみ受理する
    - 仕様上正しいタイミングで送信されたリクエストのみ受理する
  - 考慮漏れがゲームチートの原因になりやすいので、実装・設計は極力シンプルにする
    - 考慮することが減ってミス（脆弱性）も減るはず！