スマホゲームセキュリティと White Box診断のススメ

2017/02/08

グリー株式会社 セキュリティ部
脆弱性診断チーム マネージャー
山村憲和

GREE

自己紹介

山村憲和（やまむら のりかず）

1998年富士通株式会社入社

・2001年からセキュリティ部門

・主に海外対応

2012年 グリー株式会社入社

・プロダクト脆弱性診断（主に自社プロダクト）の診断員

・診断チームのプロジェクトマネジメント

アジェンダ

1. スマホゲームセキュリティ
    - ゲームにおける攻撃とは？
    - 攻撃パターン１、２、３
2. White Box診断のススメ
    - グリーの脆弱性診断体制
    - なぜWhite Box診断なのか
    - 実際の話

1 スマホゲームセキュリティ

GREE

ゲームにおける攻撃とは？

GREE

ゲームを攻撃する動機

- レアカードやレアアイテムをGETしたい
- 人より優位に立ちたい
- 強くなりたい
- お金を掛けたくない
- そして楽して簡単に達成したい

チートが最も多い攻撃

攻撃パターン1、2、3

攻撃パターン１、２、３全体像

GREE

パターン１
Webゲーム時代

リクエストを攻撃

サーバー

内部データを攻撃

パターン３
Nativeゲーム時代

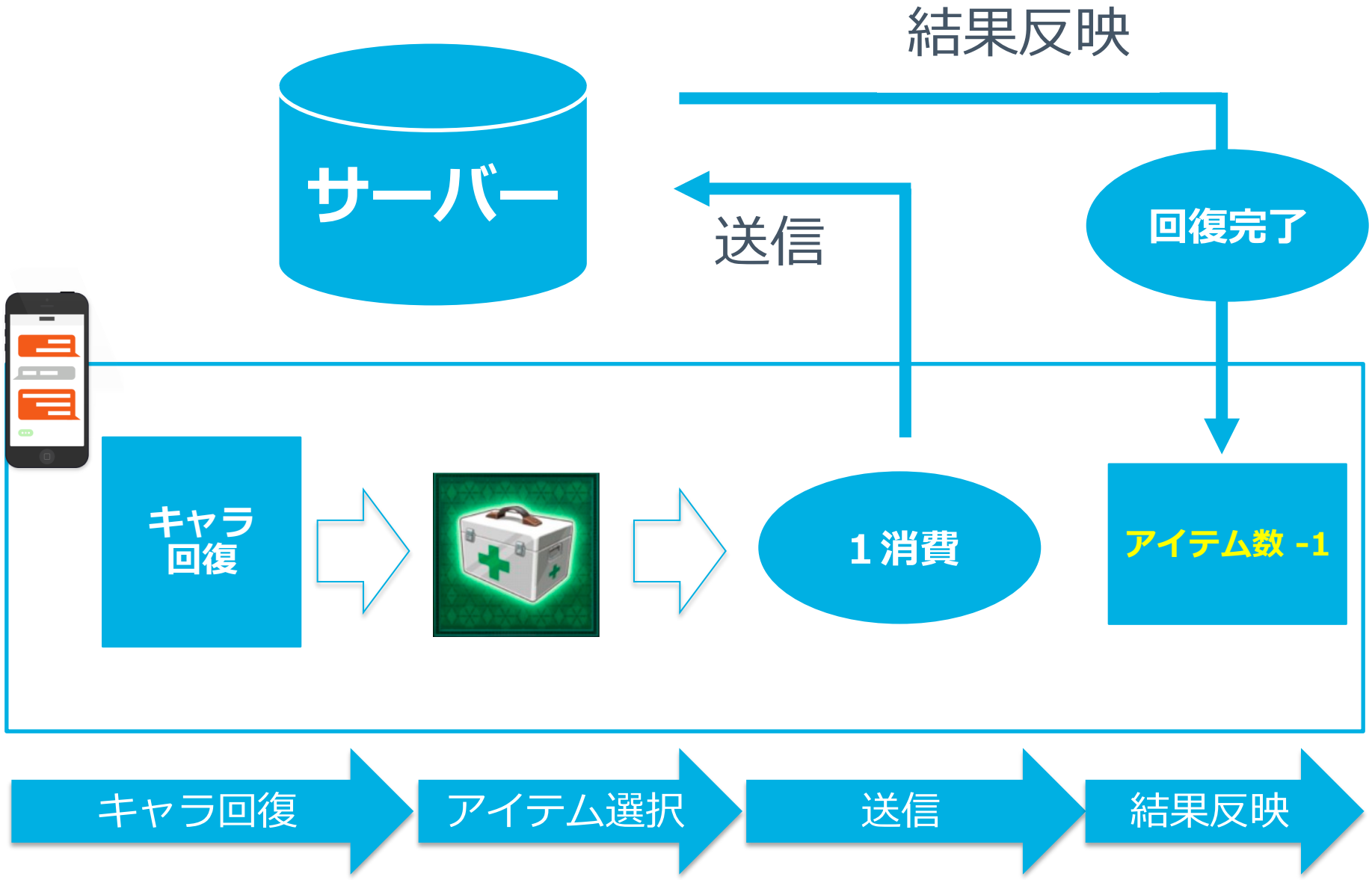
レスポンスを攻撃

パターン２
Web ～ Native
ゲーム移行期

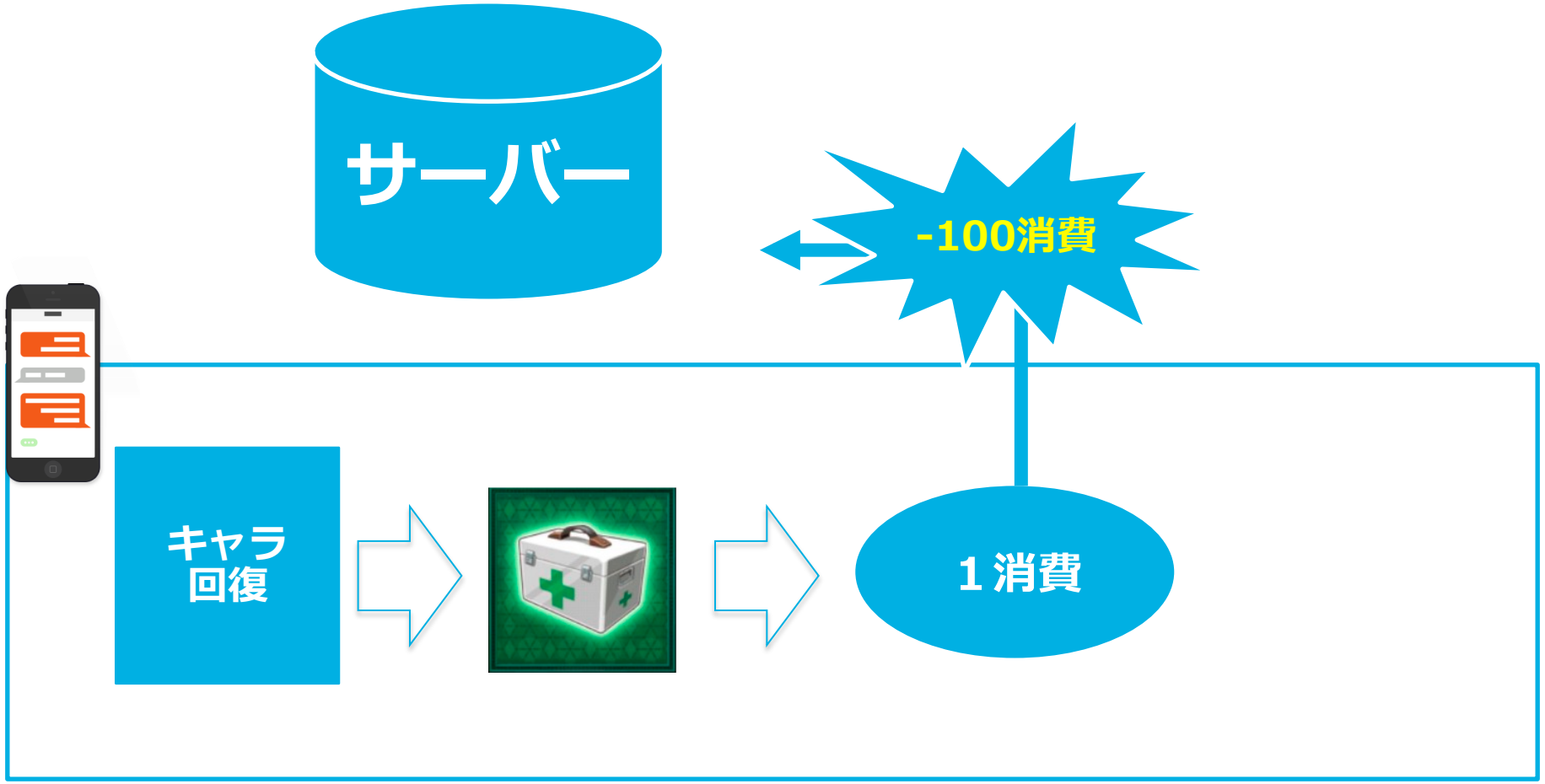
パターン１：リクエストの改ざん

- 勝敗結果、攻撃力、防御力、アイテム数
- 楽して簡単に目的を達成する
- 効果的、手間が最小限
- 生産性が高く魅力的
- ここをまずは狙う

パターン１：リクエストの改ざん

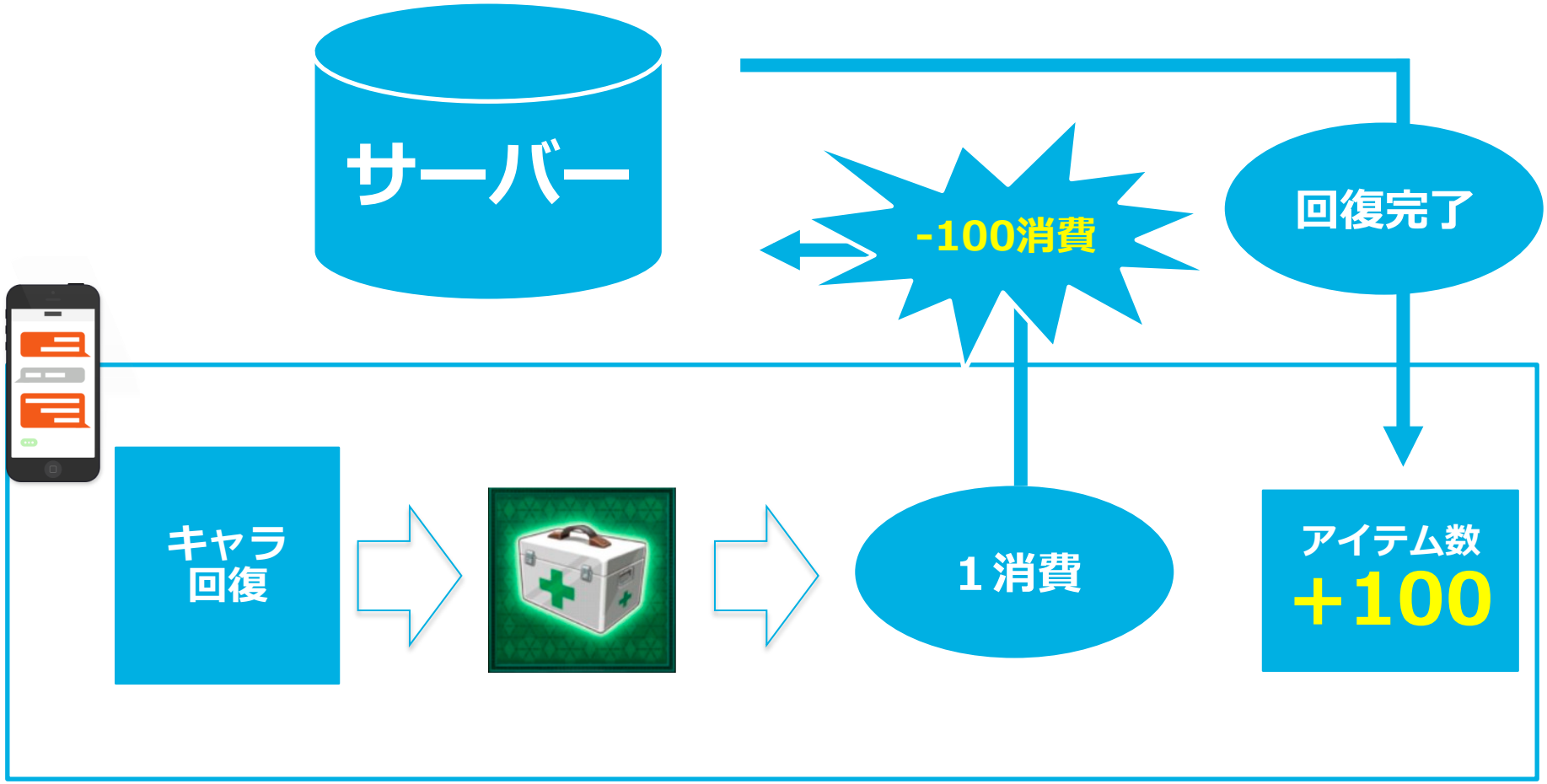


パターン１：リクエストの改ざん



キャラ回復 → アイテム選択 → 結果送信 → 結果反映

パターン１：リクエストの改ざん



キャラ回復 → アイテム選択 → 結果送信 → 結果反映

パターン１：リクエストの改ざん

```
{
  u'ctag': None,
  u'payload':
        {
                u'dataCharacterId': 632,
                u'dataItemId': 399,
                u'dataItemNum': 1
        },
  u'sessionId': None,
  u'uuid': None
}
```

```
{
  u'ctag': None,
  u'payload':
        {
                u'dataCharacterId': 632,
                u'dataItemId': 399,
                u'dataItemNum': -100
        },
  u'sessionId': None,
  u'uuid': None
}
```

パターン１：リクエストの改ざん

```
//リクエストパラメータの取得
$dataItemNum = $this->getRequest()->getParameter('dataItemNum', 0);
```

```
//アイテム数の減算処理
$this->gameClientServiceItem ->insertDataItem($userId,
$dataItem['masterItemId"], -$dataItemNum);
```

```
//バリデーターでマイナス値が排除されていない
$validateRules[] = array('dataItemNum", 'required", 'int");
```

ステップ１：リクエストの改ざん

- 効果的、手間が最小限
  - パラメータを変えるだけ
  - プロキシツール使うだけ
- 生産性が高く魅力的
  - アイテムを地道に増やすことなく
    一気に集められる。
  - 自分のキャラをいくらでも回復できる

ブラウザゲーム主流のころによくあったパターン

ステップ１：リクエストの改ざん

- サーバー側でしっかり対策する
- データを暗号化してても怠らない
- データの正当性の確認
- 想定外のデータの排除

ステップ 1 : リクエストの改ざん

```
//バリデーターでマイナス値を排除する
$validateRules[] = array('dataItemNum', 'int', 'min' => 1);
```

```
// アイテムデータ所持数の確認
if ($dataItemNum > $dataItem['num']) {
    ExceptionUtil::ClientException(
        ErrorMessage::ITEM_ENHANCE_ERROR,
        __METHOD__,
        sprintf('DataItem Num Short userId = %s, dataItemId = %s, %s
> %s', $userId, $dataItemId, $dataItemNum, $dataItem['num'])
    );
} elseif ($dataItemNum < 1) {//マイナス値排除

    ExceptionUtil::ClientException(
        ErrorMessage::ITEM_ENHANCE_ERROR,
        __METHOD__,
        sprintf('Invalid DataItem Num: userId = %s, dataItemId = %s,
%s', $userId, $dataItemId, $dataItemNum)
    );
}
```

パターン１：リクエストの改ざん

```
//ログに記録し監視に活用する
$this->gameClientServiceCharacter->insertLogCharacterEnhance(
            $userId,
            $dataCharacterId,
            $dataItemId,
            $dataItem['masterItemId'],
            $dataItem['num'],
            $dataItem['num'] - $dataItemNum,
            $dataCharacterBefore['CharacterLevel'],
            $dataCharacterAfter['CharacterLevel'],
            $dataCharacterBefore['CharacterExp'],
            $dataCharacterAfter['CharacterExp']
        );
```

さらに

GREE

パターン１：リクエストの改ざん

リクエストデータの改ざん検知の仕組みを導入

```
{"ctag":None,"payload":{"dataCharacterId”:632,"dataItemId":
399,"dataItemNum": 1},"sessionId":None,"uuid":None}

=> 8cc7277ba0a9c8bf4cfd5420428302a8ddba0f84
```

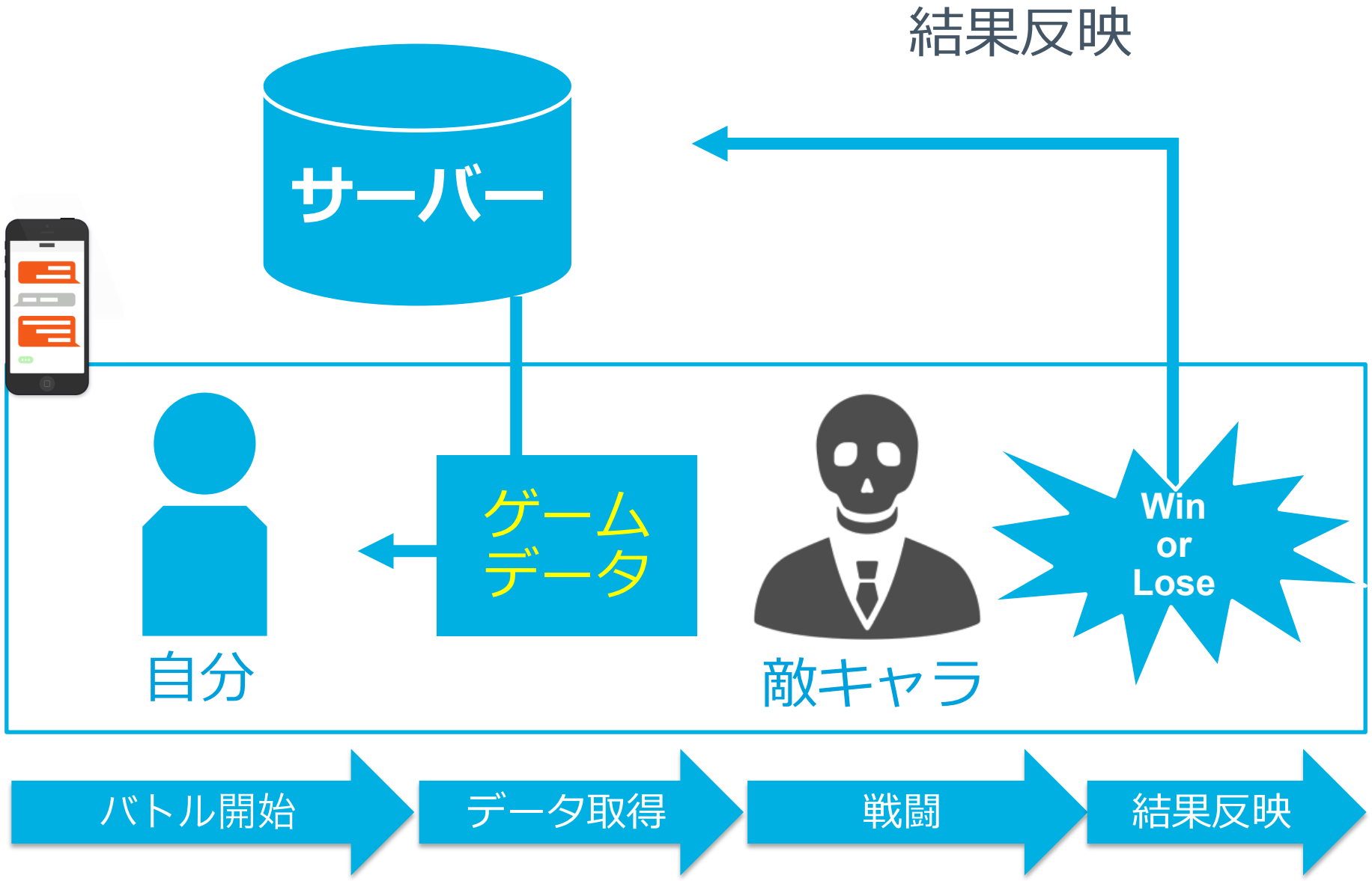
```
{"ctag":None,"payload":{"dataCharacterId”:632,"dataItemId":
399,"dataItemNum": -100},"sessionId":None,"uuid":None}

=> caf64fea1ac722b0785bba0b182f9ad22f950d92
```

- リクエストデータやその他の情報をベースにハッシュを生成、ヘッダにつけてサーバへ送信
- 改ざんしても、サーバーでの検証で改ざんが検知できる。
- 攻略されてしまう現状はあるが、やらないよりはマシ。

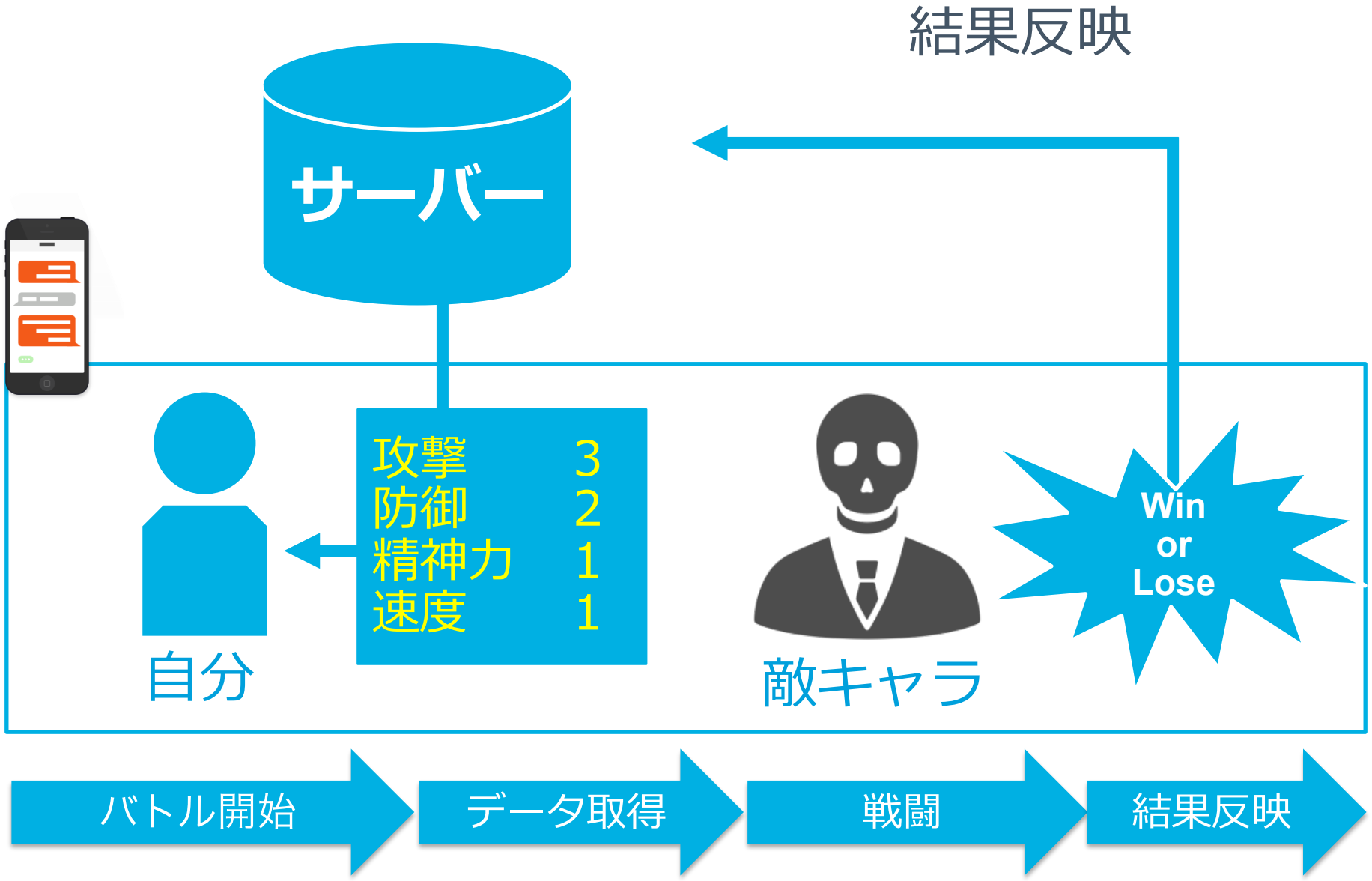
パターン２：レスポンス改ざん

- ステップ１の攻撃が対策されている場合
- クライアントに送られる攻撃力、防御力など
- スマホゲーム（ネイティブ）では効果的
- 楽して簡単に目的を達成する
- 効果的、手間が最小限
- 生産性が高く魅力的

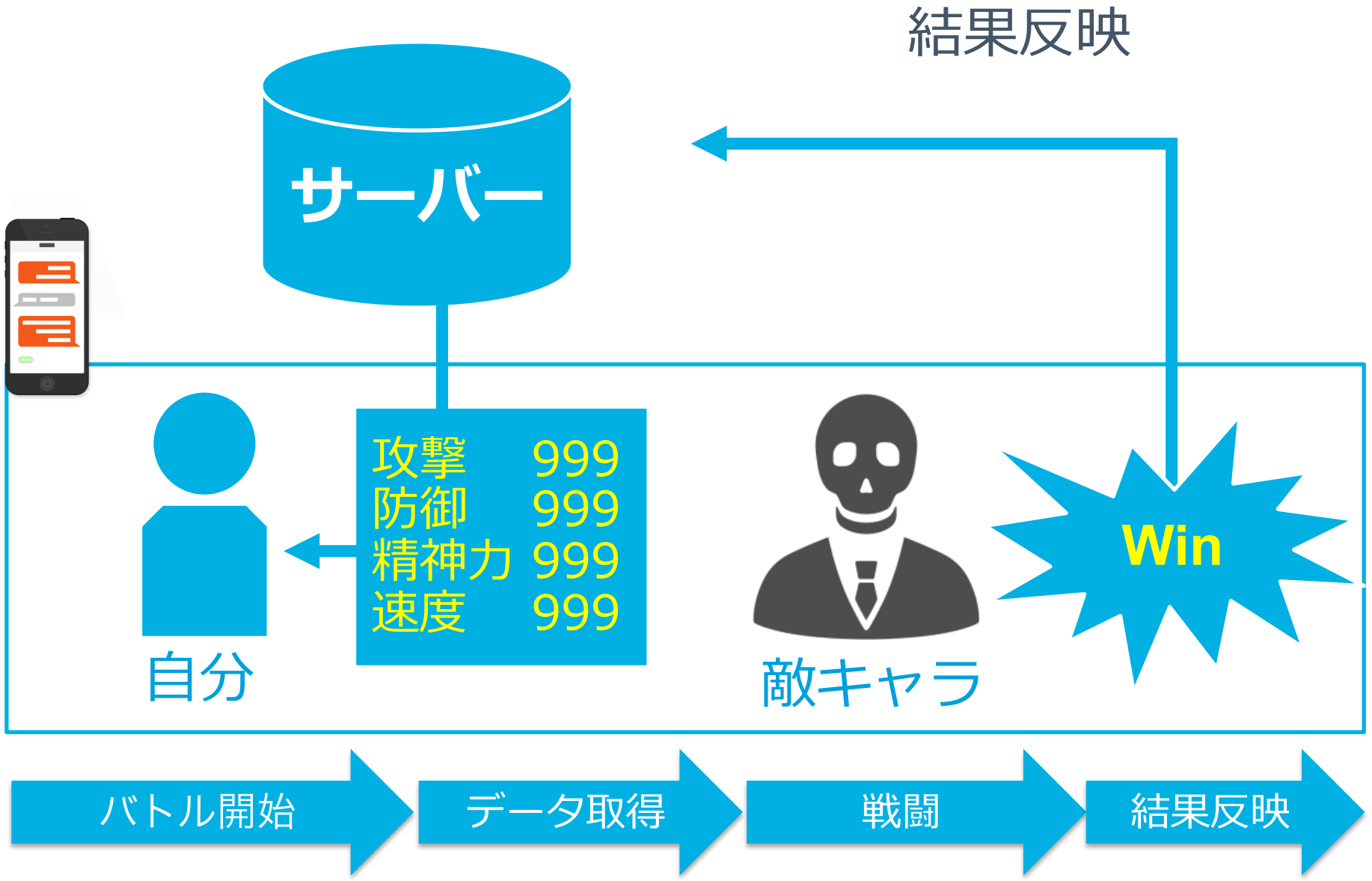
パターン2: レスポンスの改ざん



パターン2: レスポンスの改ざん



パターン2: レスポンスの改ざん



パターン2: レスポンスの改ざん

```
{"characters":
   {
     "id": "1",
     "master_character_id": "10001",
     "level": "2",
     "exp": "194",
     "hp": "999",
     "mp": "999",
     "atk": "999",
     "def": "999",
     "mentality": "999",
     ”speed": "999",
     ”aspeed": "999"
   }
 }
```

パターン２：レスポンスの改ざん

- 効果的、手間が最小限
  - パラメータを変えるだけ
  - プロキシツール使うだけ
- 生産性が高く魅力的
  - 無敵になれる
  - 次々とバトルをクリアできる

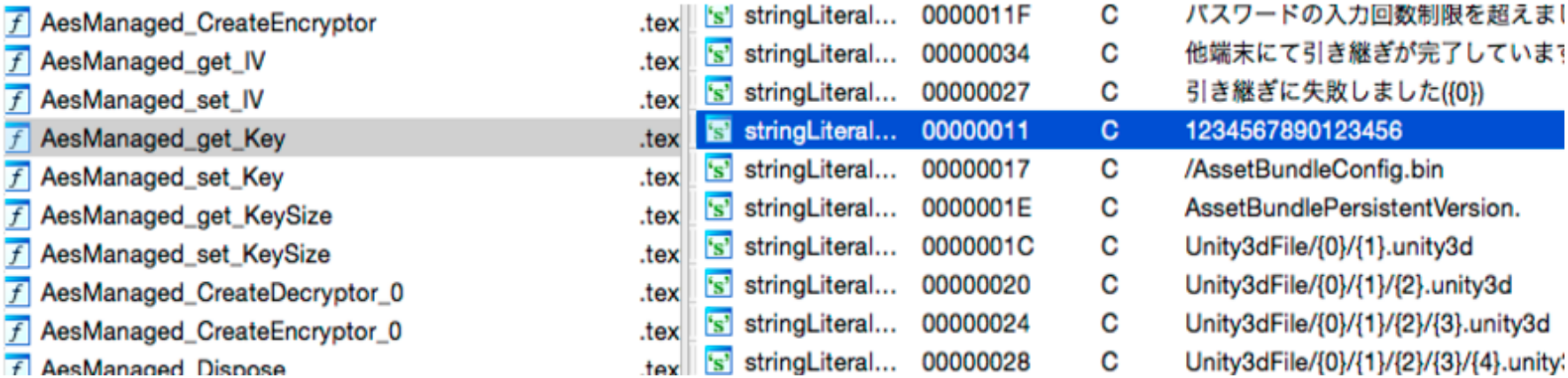
ネイティブゲームに見られるパターン

パターン２：レスポンスの改ざん

- 通信データの改ざんを検知（ハッシュ）
- 通信データの暗号化
- ただし突破されると思っておいたほうが良い
- 各種行動ログの取得
- 異常値の監視
- など

パターン２：レスポンスの改ざん

- 暗号化（例えばAES）
  - 鍵が抜かれたら通信データは見れる
- ハッシュによる改ざん検出
  - ハッシュ生成前に改ざんしてしまう
  - クライアントのチェックロジックを消す



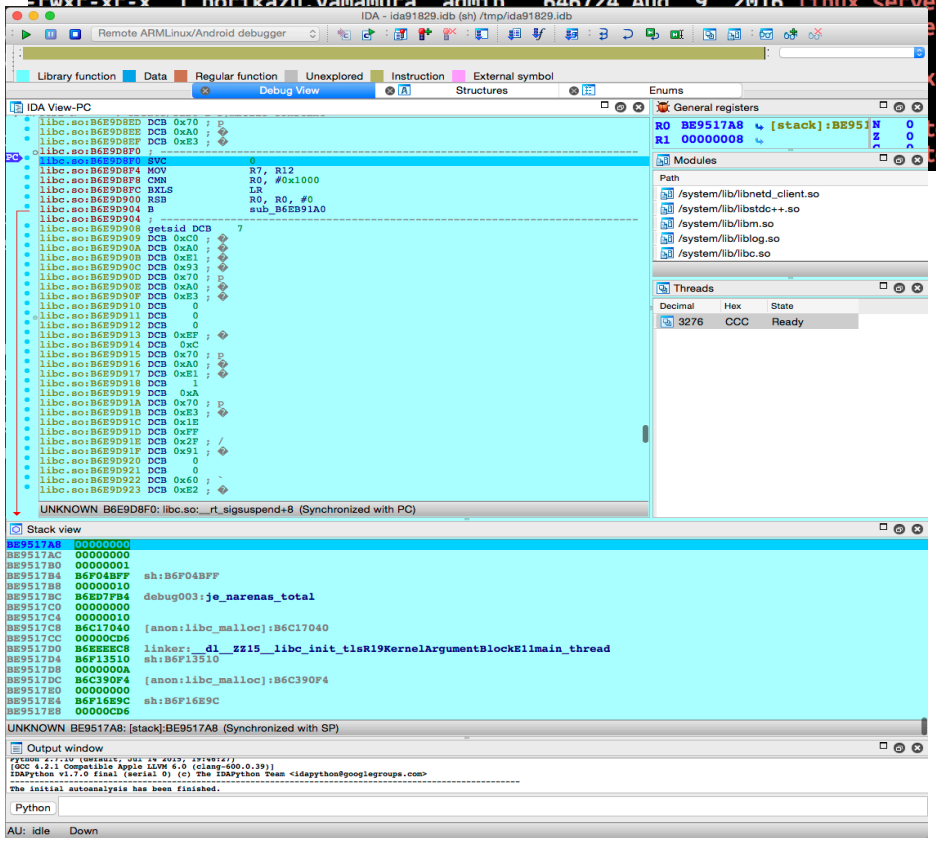
パターン３：内部データの改ざん

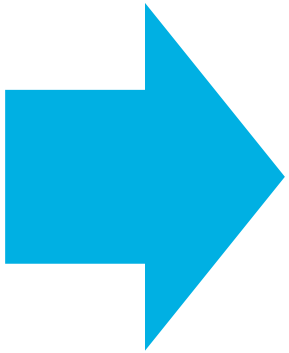
- リバースエンジニアリング
- メモリ改ざん、バイナリ改ざん
- パターン１、２よりは敷居が高い
- 選択肢として優先度は低い
- が、できれば基本何でも可能
- ここから攻める人ももちろんいる
- 簡単にしてくれるツールも多々あり

パターン３：内部データの改ざん

デバッガ

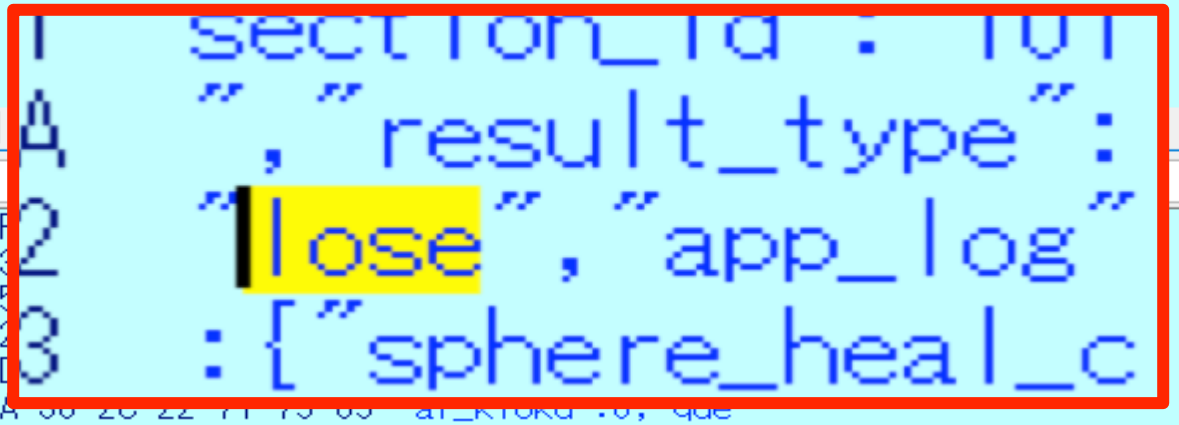
```
-rwxr-xr-x  1 norikazu.yamamura  admin   539864 Aug  9  2016 android_server
-rwxr-xr-x  1 norikazu.yamamura  admin  1152480 Aug  9  2016 android_server64
-rwxr-xr-x  1 norikazu.yamamura  admin   527576 Aug  9  2016 android_server_nonpie
-rwxr-xr-x  1 norikazu.yamamura  admin  1188112 Aug  9  2016 android_x64_server
-rwxr-xr-x  1 norikazu.yamamura  admin   850444 Aug  9  2016 android_x86_server
-rwxr-xr-x  1 norikazu.yamamura  admin   713436 Aug  9  2016 armlinux_server
-rwxr-xr-x  1 norikazu.yamamura  admin   944176 Aug  9  2016 armuclinux_server
-rwxr-xr-x  1 norikazu.yamamura  admin  1213200 Aug  9  2016 dbghelp.dll
-rwxr-xr-x  1 norikazu.yamamura  admin   646724 Aug  9  2016 linux_server
```





パターン３：内部データの改ざん

```
MEMORY:96B9B6BD DCB 0x65 ; e
MEMORY:96B9B6BE DCB 0x22 ; "
MEMORY:96B9B6BF DCB 0x3A ; :
MEMORY:96B9B6C0 DCB 0x22 ; "
MEMORY:96B9B6C1 DCB 0x6C ; l
MEMORY:96B9B6C2 DCB 0x6F ; o
MEMORY:96B9B6C3 DCB 0x73 ; s
MEMORY:96B9B6C4 DCB 0x65 ; e
MEMORY:96B9B6C5 DCB 0x22 ; "
MEMORY:96B9B6C6 DCB 0x2C ; ,
MEMORY:96B9B6C7 DCB 0x22 ; "
MEMORY:96B9B6C8 DCB 0x61 ; a
UNKNOWN 96B9B6C1: MEMORY:96B9B6C1
```



Hex View-1

```
9B630  30 31 35 32 22 2C 22 64  72 6F                    ar_kioka :0, que
9B640  22 3A 6E 75 6C 6C 2C 22  73 65                    st_id":"10102","
9B650  22 75 6E 69 74 5F 65 78  70 65                    section_id":"101
9B660  22 3A 33 36 2C 22 75 73  65 72                    ","result_type":
9B670  69 65 6E 63 65 22 3A 33  36 7D                    "lose","app_log"
9B680  61 72 5F 6B 69 6F 6B 75  22 3A                    :{"sphere_heal_c
9B690  73 74 5F 69 64 22 3A 22  31 30 31 30 32 22 2C 22  ount":0,"wait_sk
9B6A0  73 65 63 74 69 6F 6E 5F  69 64 22 3A 22 31 30 31  ill_master_ids":
9B6B0  22 2C 22 72 65 73 75 6C  74 5F 74 79 70 65 22 3A  [],"sphere_dark_
9B6C0  22 6C 6F 73 65 22 2C 22  61 70 70 5F 6C 6F 67 22  count":0,"mode_c
9B6D0  3A 7B 22 73 70 68 65 72  65 5F 68 65 61 6C 5F 63  hange_count":0,"
9B6E0  6F 75 6E 74 22 3A 30 2C  22 77 61 69 74 5F 73 6B  trap_count":0,"s
9B6F0  69 6C 6C 5F 6D 61 73 74  65 72 5F 69 64 73 22 3A  phere_water_coun
9B700  5B 5D 2C 22 73 70 68 65  72 65 5F 64 61 72 6B 5F  t":0,"item_invin
9B710  63 6F 75 6E 74 22 3A 30  2C 22 6D 6F 64 65 5F 63  cible_count":0,"
9B720  68 61 6E 67 65 5F 63 6F  75 6E 74 22 3A 30 2C 22  retry_count":0,"
9B730  74 72 61 70 5F 63 6F 75  6E 74 22 3A 30 2C 22 73  fall_count":0,"c
9B740  70 68 65 72 65 5F 77 61  74 65 72 5F 63 6F 75 6E  lear_time":0,"ma
9B750  74 22 3A 30 2C 22 69 74  65 6D 5F 69 6E 76 69 6E  x_chain_count":0
9B760  63 69 62 6C 65 5F 63 6F  75 6E 74 22 3A 30 2C 22  ,"item_flight_co
9B770  72 65 74 72 79 5F 63 6F  75 6E 74 22 3A 30 2C 22
9B780  66 61 6C 6C 5F 63 6F 75  6E 74 22 3A 30 2C 22 63
9B790  6C 65 61 72 5F 74 69 6D  65 22 3A 30 2C 22 6D 61
9B7A0  78 5F 63 68 61 69 6E 5F  63 6F 75 6E 74 22 3A 30
9B7B0  2C 22 69 74 65 6D 5F 66  6C 69 67 68 74 5F 63 6F
```

パターン３：内部データの改ざん

- 低レイヤーの知識がある程度必要
  - メモリ上にある値を操作
  - 通信の暗号化、改ざん検出は役に立たない
- クライアントを完全コントロール
  - データ操作
  - ロジック操作

ネイティブゲームに見られるパターン

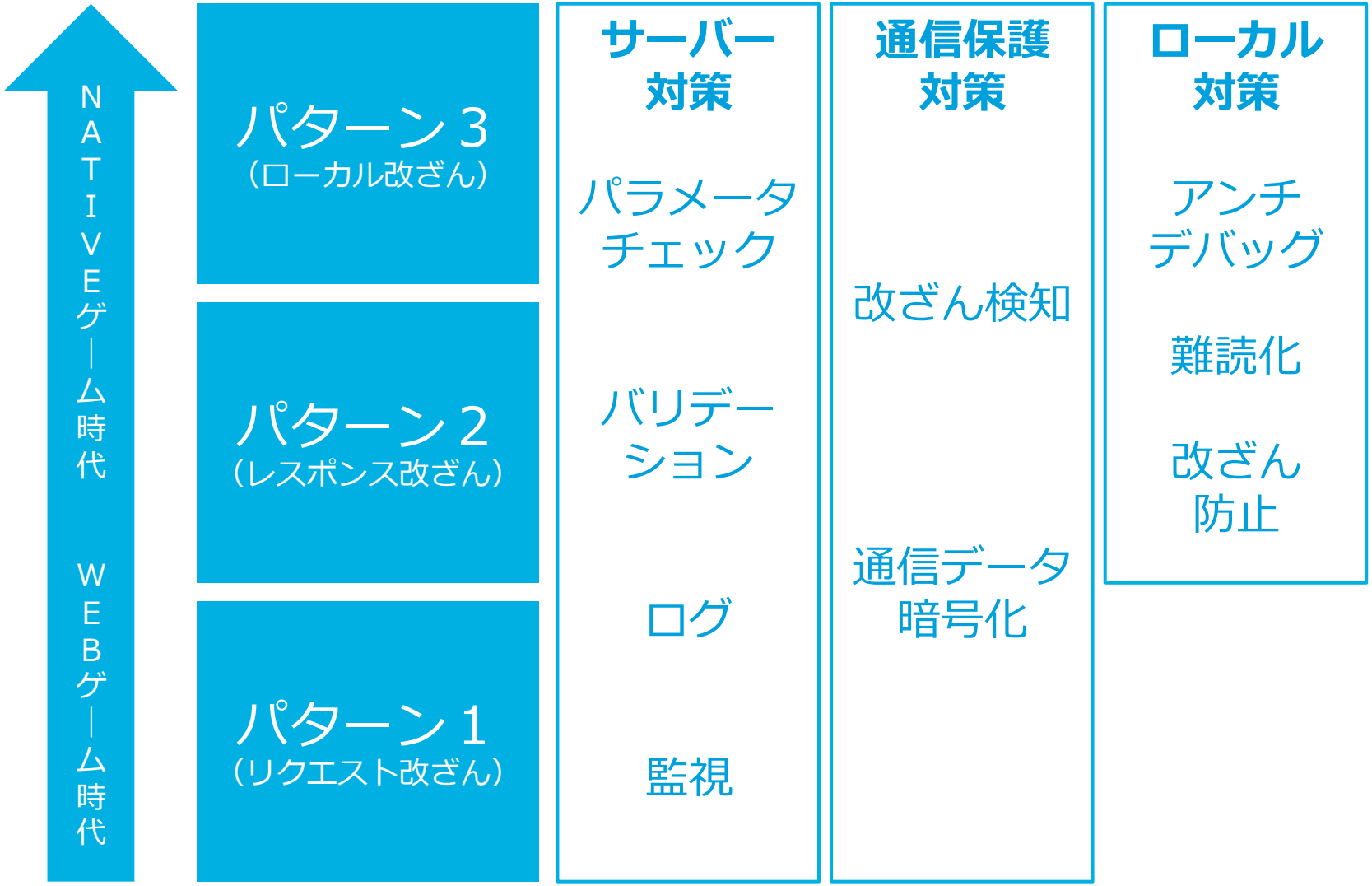
パターン３：内部データの改ざん

- 安全にするのは難しい
  - イタチごっこにならざるをえない
  - 対策できることはやった方がいい
  - 難読化、アンチデバッギング等
  - 専用のソリューション（コストとの兼ね合い）
- 被害が大きいところを見極め、ローカルで処理しない
  - コイン消費等、被害が大きい機能は必ずサーバーサイドで処理
  - サーバーかローカルか、見極めとバランス

まとめ

GREE

完璧を求めない。対策を組合わせ難しくする。GREE



2 White Box診断のススメ

グリーの脆弱性診断体制

診断メニュー	診断数（年間）	工数/１プロダクト
内製脆弱性診断 *内部リソースで診断	約２０プロダクト	Web: 10営業日 Native: 15営業日
外注脆弱性診断 *セキュリティベンダへ診断を発注	約２０プロダクト	Web: 10営業日 Native: 15営業日

- **内製脆弱性診断**
  - **ソースコードレビューによるWhite box診断**
  - **３名体制**
  - **新規リリースプロダクトが多い**
  - **診断調整 → 診断 → レポート → 修正トラッキング**

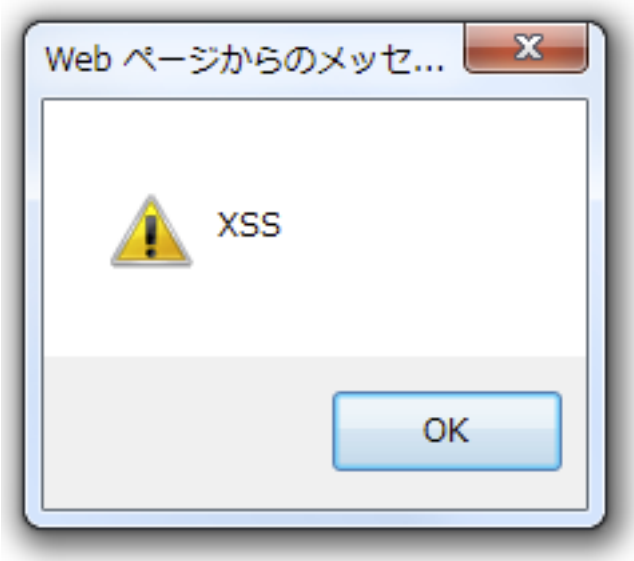
なぜWhite Box診断なのか

- メリット
  - 効率的
  - コードが見れる（全部書いてある）
  - どこが悪いのかを指摘しやすい
  - 潜在的な脆弱性を発見できる
  - 開発側と連携がとりやすい

実際の話

GREE

みんな大好きXSS



皆さんどんなフィードバックしてます？

「安全なウェブサイトの作り方」からいくつか参考として抜粋

- ウェブページに出力する全ての要素に対して、エスケープ処理を施す。
- URL を出力するときは、「http://」や 「https://」で始まる URL のみを許可する。
- スタイルシートを任意のサイトから取り込めるようにしない。

私たちはこんな感じに診断して
フィードバックしています

GREE

診断時の着眼点

- 開発フレームワークのセキュリティ機能、実装の確認、理解
- XSS対策機能の設定が正しいか？
- その対策機能が適切に使われているか？
- 対策機能をバイパスするように無理くりいじられていないか？
- XSSを作りこみやすい仕組み（設定、コーディング）に
  なっていないか？
- 今問題なくても、将来的にXSSが発動する可能性はないか？

FuelPHP – 1.7

GREE

出力エンコーディング

デフォルトでは、Fuel は入力フィルタリングより出力エンコーディングを好みます。その理由は2つの要素からなります。 あなたのデータがどこから来たものであろうと、そしてそれがフィルタされていようがいまいが、出力エンコーディングはその**データを クライアントに送信する際に無害にします。**これは、すべての入力は生のまま、変更のないかたちで保存され、何が起ころうと、あなたはいつでも元データにアクセスできるとことも意味します。

セキュリティ設定 – config.php

```
//デフォルト（fuel/core/config/config.php）
'auto_filter_output"  => true,

//デフォルト（fuel/app/config/config.php）
//'auto_filter_output'  => true,
```

```
//app設定フィルタ(fuel/app/config/config.php)のデフォルト設定値。
'output_filter' => array('Security::htmlentities');
```

```
//以下のようにoutput_filter自体をコメントアウトしても許可されておらずエラーになる。(Securityクラスの_init()でチェックされている)
//'output_filter' => array('Security::htmlentities');

//こんなことすれば無効化はできる。万一やっていたら経緯の把握と危険性を議論。
'output_filter' => array();
```

coreのレベルで設定値が有効となっているぞ。
htmlentitiesが使われているぞ。

実装の部分（view.php）

```
public static function forge($file = null, $data = null,
$auto_filter = null)
  {
    return new static($file, $data, $auto_filter);
  }
```

```
public function __construct($file = null, $data = null,
$filter = null)
{
    - snip -
    $this->auto_filter = is_null($filter) ?
\Config::get('security.auto_filter_output', true) : $filter;
    - snip -
}
```

$filterを明示的にfalseにしていないかな？
configの設定値はtrueだから大丈夫そうかな？

実装の部分（外部テンプレートエンジン）

GREE

```
class View_Mustache extends \Parser\View_Mustache
{
    public static function forge($file = null, $data = null,
$auto_filter = false)
     {
```

ここで無効になってた。。。。＞＜;

```
<div class="Article-body">
        <div class="{{#mustache_fns.ucwords}}{{content_type}}{{/
mustache_fns.ucwords}}-content">
          <h1 class="Article-title">{{{content.title}}}</h1>
          {{#content.description}}<p class="{{#mustache_fns.ucwords}}
{{content_type}}{{/mustache_fns.ucwords}}-teaser">{{{content.description}}}</p>{{/
content.description}}
```

一斉摘発

$> grep ‘{{{‘ /hoge/fuga/<tplのディレクトリ>

最終的にこんなフィードバックになる

GREE

- フレームワークレベルでは対策が無効化されてますね
- テンプレートエンジンのエスケープ機能に依存してますね
- フレームワークかテンプレートエンジンのどっちかに寄せたかったんですよね？（二重エスケープがイヤだった？）
- でも、テンプレート書く人がこの実装、ルールを理解しておかないと漏れが出ないですかね？
- そのあたり開発ルールや、チェック体制を整備しておかないと将来的にxssを作り込みそう

どうしましょっか？大丈夫です？（にこやかに）

実装を理解し、コードの行間を読み、潜在的な脆弱性にも気を配る努力

まとめ

- 実装を理解することで効率的に脆弱性を検知できる
- モノによっては一斉摘発可能
- コードレベルで開発部門と対策を練ることができる
- 具体的なので開発部門もうれしい (^_^)
- 技術ベースで事業に貢献する（弊社開発本部の目標）
- ユーザーの皆様へ安全な弊社サービスをお届けする！
- そして、、、、、、、、、

インターネットを通じて、
世界をより良くする。