Finding vBulletin 0-days through poor man's symbolic execution

Charles Fol
folcharles@gmail.com

Lexfo

Résumé. Avec plus de cent mille installations, vBulletin est le logiciel communautaire leader du marché. Ce CMS de forum en ligne, développé en PHP, est utilisé par des organisations telles que Steam, EA, Sony, ou même la NASA. L'année dernière, l'outil a été la cible d'une vulnérabilité 0-day permettant l'exécution de code sans authentification. Plus généralement, le produit, qui existe depuis 2000, a subit au cours des années de nombreuses attaques, notamment des injections SQL. Après avoir procédé a un audit de sécurité du logiciel, de nombreuses failles furent découvertes. Je présente ici la méthode que j'ai utilisée pour vérifier la sécurité des requêtes SQL, sujet massif (plusieurs centaines de requêtes dynamiques) et ténu du logiciel (10 CVEs en 12 ans), en explorant automatiquement les différentes ramifications permettant de contrôler partiellement des requêtes SQL, et ayant permis de détecter une vulnérabilité pré-authentification. Je présente ensuite l'exploitation de cette dernière afin de prendre le contrôle d'un compte administrateur et ainsi d'exécuter du code sur le serveur.

1 Introduction

1.1 Le produit

vBulletin est un logiciel internet de forum propriétaire, vendu par MH Sub I, LLC. Il est codé en PHP et utilise une base de données MySQL. La solution est décrite comme « the world's leading community software » (le logiciel communautaire leader), et affiche plus de 100,000 installations dans le monde. Il est utilisé par de nombreuses organisations telles que Steam, EA, Sony, ou même la NASA. Il existe depuis 2000, est à l'heure où ces lignes sont écrites à la version 5.6.0, et est nommé depuis sa version 5 vBulletin 5 Connect.

1.2 (In)sécurité

Au travers des années, de nombreuses vulnérabilités critiques ont été découvertes sur le produit. On compte notamment les injections

SQL comme le type « phare » de vulnérabilité, avec 10 CVEs en 12 ans [7]. En 2019, la sécurité de vBulletin a encore été ébranlée lorsqu'une vulnérabilité, pré-authentification, nécessitant une seule requête HTTP, et permettant d'exécuter du code, a été révélée (CVE-2019-16759) [6]. La vulnérabilité, triviale, semblait pour beaucoup avoir été introduite volontairement (« backdoor »). L'intervention du CEO de Zerodium sur Twitter, ou il affirme aussi avoir connaissance de la vulnérabilité depuis trois ans, allait dans ce sens [4].

1.3 Tainting et exécution symbolique

On se focalise ici sur la recherche d'injections SQL exploitables avant authentification, en se concentrant sur l'API du produit. L'idée, courante pour ces vulnérabilités, est de trouver un paramètre fourni par l'utilisateur qui est utilisé dans une requête SQL, et pas suffisamment échappé ; la difficulté de l'exercice vient du nombre d'API disponibles, de librairies présentes, et de requêtes SQL dynamiques différentes utilisées par le produit (plusieurs centaines). Afin de m'épargner la tâche de parcourir le code à la main, j'ai conçu un outil qui permet de savoir quels paramètres utilisateurs sont utilisés pour construire des requêtes SQL, et s'ils sont suffisamment contrôlés et sécurisés.

A travers cette présentation, je vais décrire l'architecture globale du produit côté front-end, puis montrer comment j'ai utilisé du tainting de variables et de l'exécution symbolique pour vérifier quels paramètres utilisateur peuvent être utilisés dans des requêtes SQL, et permettre une injection.

Enfin, je montrerai comment convertir l'injection SQL trouvée en RCE, par une escalade de privilège suivie d'une exécution de code.

2 Architecture du produit

2.1 API

La plus grande partie de l'interaction de vBulletin avec un utilisateur standard est effectuée via l'**API**. Celle-ci est implémentée via des classes ayant pour préfixe `vB_Api_`. Chaque méthode de ces classes peut être appelée via une requête HTTP, et ses arguments sont obtenus à partir des paramètres GET ou POST de la requête.

Par exemple, la méthode `fetchProfileInfo($userid)` de la classe `vB_Api_User` s'invoque en utilisant l'URL suivante : `/ajax/api/user/fetchProfileInfo?userid=3`

Pour manipuler les données envoyées, l'**API** utilise des **librairies**.

2.2 Librairies

Dans vBulletin, une **librairie** existe pour chaque catégorie de données. Ainsi, une librairie existe pour gérer les utilisateurs (`User`), les photos (`Content_Photo`), les pièces jointes (`Content_Attachment`), etc. Comme pour l'API, chaque librairie est implémentée sous forme d'une classe PHP, cette fois préfixée par `vB_Library`. Les méthodes de ces librairies permettent d'obtenir, créer, ou modifier ces données. Par exemple, la méthode `fetchModerator($userid)` de la classe `vB_Library_User` retourne les permissions de modération d'un utilisateur.

Afin de lire et stocker les données qu'elles traitent, les **librairies** utilisent les **QueryDefs**.

2.3 QueryDefs

Les **QueryDefs** sont une pléthore de méthodes qui ont pour but de construire une requête SQL à partir des données envoyées, de l'exécuter, puis de retourner le résultat. Comme, par défaut, tout est stocké dans la base de données dans vBulletin, même les pièces jointes, le nombre de requêtes nécessaires est immense, et leurs utilisations nombreuses. Les classes contenant les méthodes susdites sont suffixées par `_QueryDefs`.

```
public function userSearchRegisterIP($params, $db, $check_only =
    false)
{
  if ($check_only)
  {
    return !empty($params['ipaddress']) AND isset($params['
        prevuserid']);
  }
  else
  {
    $params = vB::getCleaner()->cleanArray($params, array(
      'ipaddress' => vB_Cleaner::TYPE_NOCLEAN, //cleaned right after
          this
      'prevuserid' => vB_Cleaner::TYPE_UINT,
    ));

    if (substr($params['ipaddress'], -1) == '.' OR substr_count(
        $params['ipaddress'], '.') < 3)
    {
      // ends in a dot OR less than 3 dots in IP -> partial search
      $ipaddress_match = "ipaddress LIKE '" . $db->
          escape_string_like($params['ipaddress']) . "%'";
    }
    else
    {
      // exact match
      $ipaddress_match = "ipaddress = '" . $db->escape_string(
          $params['ipaddress']) . "'";
```

```
    }

    $sql = "
      SELECT userid, username, ipaddress
      FROM " . TABLE_PREFIX . "user AS user
      WHERE $ipaddress_match AND
        ipaddress <> '' AND
        userid <> $params[prevuserid]
      ORDER BY username
    ";

    $resultclass = 'vB_dB_' . $this->db_type . '_result';
    $result = new $resultclass($db, $sql);
    return $result;
  }
}
```

Listing 1. Exemple de QueryDef de la classe vB_dB_MYSQL_QueryDefs

Une liste de paramètres est fournie en premier argument de la méthode : ils sont utilisés pour construire une requête SQL, avant de l'exécuter. Chaque méthode **QueryDef** sera successivement appelée avec le troisième paramètre (le booléen `$check_only`), à `true`, puis à `false`. Lors du premier appel, la méthode est chargée de vérifier que les paramètres sont bien définis, et/ou sécurisés. Si cet appel renvoie vrai, alors un second appel est fait, cette fois pour construire et exécuter la requête SQL, en sécurisant aussi les paramètres au passage. Dans l'exemple 1, le code va s'assurer lors de la première exécution que `$ipaddress` et `$prevuserid` sont bien définis. Si c'est le cas, alors la requête permettant de lister les utilisateurs ayant une addresse IP commencant par `$ipaddress` sera construite et exécutée.

2.4 Architecture globale

L'architecture d'API du produit pour un utilisateur standard (non administrateur) est donc la suivante (image 1) :

— L'API, recevant les paramètres utilisateurs ;
— Les librairies, étape intermédiaire ;
— Les QueryDefs, qui vérifient des paramètres puis exécutent des requêtes SQL.

En termes de chiffres, on dénombre plus de **1000 méthodes d'API**, **850 méthodes de librairies**, et environ **250 QueryDefs**.

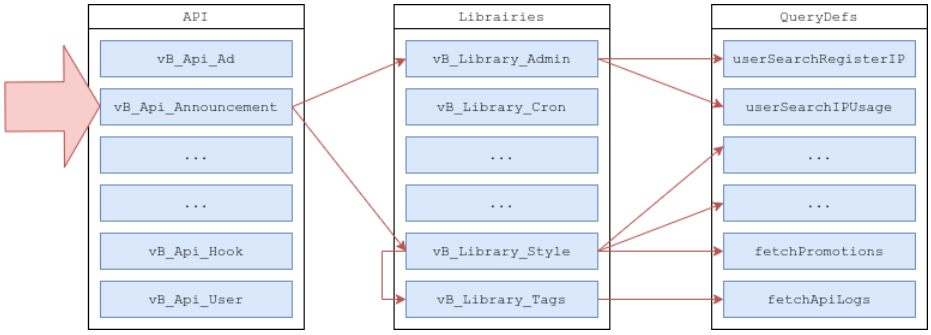

Fig. 1. Une requête HTTP et les interactions qu'elle produit entre les différents blocs

3 Sécurité des requêtes SQL

3.1 Construction

Comme on peut le voir sur l'exemple de code 1, les requêtes SQL sont construites « à la main », en concaténant des chaines de caractères. Le code permettant de les construire est parfois simple (comme dans l'exemple) mais il est parfois plus complexe, utilisant jusqu'à 10 variables, et de nombreuses conditions et boucles. Les paramètres sont soit filtrés au préalable (classe `vBCleaner`, cast vers `int`, etc.), soit échappés lors de leur concaténation dans la requête (`$db->escape_string()`). Bien que la majorité du filtrage (échappement des caractères dangereux SQL) soit effectuée dans les QueryDefs, les API et librairies limitent généralement le type et le nom des paramètres d'entrée à ces fonctions. En d'autre termes, avant d'être utilisé dans une requête SQL, un paramètre utilisateur peut être filtré ou vérifié à chaque étape de son parcours. Par exemple, l'API peut vérifier qu'un paramètre est un tableau, avant de l'envoyer à une librairie, qui vérifiera que ce paramètre contient simplement des entiers, avant finalement de l'envoyer à une QueryDef afin qu'il soit utilisé dans une requête SQL. De la découle la complexité de la tâche : afin de détecter une injection, le paramètre doit être suivi précisément. Prenons par exemple la méthode QueryDef `userReferrals` :

```
public function userReferrals($params, $db, $check_only = false)
{
  if ($check_only) return !empty($params['referrerid']);

  $where = "WHERE referrerid = " . $params['referrerid'] . " AND
      usergroupid NOT IN (3, 4)";
  if (!empty($params['startdate'])) $where .= " AND joindate >= "
      . $params['startdate'];
```

```
    if (!empty($params['enddate'])) $where .= " AND joindate <= " .
        $params['enddate'];
    if (!empty($params['enddate'])) {}
    $sql = "SELECT username, posts, userid, joindate, lastvisit,
        email
        FROM " . TABLE_PREFIX . "user
        $where
        ORDER BY joindate DESC";

    $resultclass = 'vB_dB_' . $this->db_type . '_result';
    $result = new $resultclass($db, $sql);
    return $result;
  }
```

Listing 2. QueryDef userReferrals (simplifiée)

Les 3 paramètres `$referrerid`, `$startdate` et `$enddate` sont utilisés directement pour construire une requête, ce qui produit 3 injections SQL évidentes. Cependant, en recherchant les points d'accès à cette méthode, on remarque tout d'abord que celle-ci n'est accessible que par le panel administrateur, et que les paramètres qui lui sont envoyés sont castés en `int` au préalable.

En conséquence, il n'y a pas d'injection SQL.

3.2 Écarter une méthode manuelle

Ce schéma, très simple, se reproduit partout entre l'API et les QueryDefs. Le code de vérification des paramètres QueryDefs, notamment, est très redondant et peu organisé. Il est donc facile de faire une erreur, tant pour le développeur que pour l'auditeur, surtout quand on considère qu'un QueryDef peut être appelé par différentes librairies, elles-même appelées par différentes API, ayant chacune appliqué leur couche de filtrage sur les paramètres.

Je souhaite donc éviter une approche manuelle, et je m'oriente donc vers de l'automatisation.

Afin de savoir quelles entrées utilisateurs atteignent les QueryDefs, j'ai choisi d'utiliser du **static taint analysis** pour suivre ceux-ci dans les API et librairies, jusqu'au appels à un QueryDef. Puis, afin de supporter la logique de filtrage et de construction plus complexe de ces dernières, j'utilise de l'**exécution symbolique**.

Le premier moteur traque, pour chaque ligne de code, quelles variables sont contrôlées par l'utilisateur. Lorsqu'une de ces variables atteint un appel à un QueryDef - généralement sous forme d'un ou plusieurs éléments du tableau `$params` - le moteur d'exécution symbolique exécute la fonction

et détermine quelles requêtes SQL sont susceptibles d'être exécutées, et si elles contiennent des paramètres utilisateurs pas suffisamment échappés.

4 Audit automatique du produit

4.1 Static Taint Analysis

Chaque paramètre d'une méthode de l'API est une source. Chaque exécution de méthode QueryDef est un puit. Pour qu'une source puisse atteindre un puit, il doit exister un chemin où elle n'est ni filtrée, ni supprimée : pour chaque embranchement possible du programme, il faut donc conserver une liste de variables qui dépendent d'une source et qui n'ont pas été sécurisées ; on les dit **contrôlées**. Les variables ne dépendant pas d'une source ou ayant été vérifiées ou sécurisées sont notées comme **non contrôlées**. Plus précisément, on cherche à garder connaissance, pour chaque ligne de code, de toutes les parties de variables qui sont construites à partir des arguments d'une méthode d'API. Lorsqu'une requête SQL sera exécutée, il faudra vérifier si elle a été influencée par une de ces variables.

On construit donc un moteur de tainting basique, et on lui fait parcourir l'AST (Arbre de la syntaxe abstraite [1]) des fonctions d'API. L'AST est extrait via la librairie PHP-Parser [3]. On maintient pour chaque bloc de code (`if-elseif-else`, `foreach`) une liste de variables contrôlées. Le fonctionnement est très basique : toutes les branches sont explorées, et aucune évaluation réelle des valeurs n'est effectuée. Lorsqu'un appel de méthode vBulletin est effectué, on change de contexte et on évalue la méthode appelée de la même manière. On peut ainsi stocker pour chaque méthode d'API, quelles QueryDefs elles vont exécuter, avec quels paramètres, et via quelles autres méthodes. La sortie de l'outil est la suivante :

```
#### QueryDef ######################
vBForum_dB_MySQL_QueryDefs::getContentTablesData
-- PARAMS --------------------------
PARAMS [not-controlled]:
  nodeid [controlled]:
-- CALL CHAINS ----------------------
:: CHAIN 0
0 vB_Api_Content::getIndexableContent
1 vB_Library_Content::getIndexableContent
2 vB_Library_Content::fillContentTableData
...
```

Listing 3. Output du moteur de tainting pour la QueryDef getContentTablesData

On voit sur le listing 3 qu'une chaine d'appels partant de la méthode d'API `vB_Api_Content::getIndexableContent` permet d'appeler le QueryDef `vBForum_dB_MySQL_QueryDefs::getContentTablesData` avec un tableau de paramètres dont on contrôle seulement une clef : `nodeid`.

4.2 Exécution symbolique

Comme énoncé précédemment, chaque appel à une méthode QueryDefs se fait en fait sous forme de deux : le premier, avec `$check_only` à `true`, afin de vérifier que les paramètres sont valides. La méthode est chargée de retourner `true` si les paramètres sont valides, ou `false` s'ils ne le sont pas. Si le premier appel retourne vrai, alors on passe au second, avec `$check_only` à `false`, qui permet de construire et d'exécuter la requête (exemple 4).

```
public function getSubscriptionUsersLog($params, $db, $check_only =
    false) {
  if ($check_only) {
    if (isset($params['sortby'])) {
      $params['sortby'] = @array_pop($params['sortby']);
      if (isset($params['sortby']['field']) AND !empty($params['
          sortby']['field']))
        if (!$this->checkSortingFields($params['sortby']['field']))
          return false;
    }

    if (isset($params[vB_dB_Query::PARAM_LIMIT]) AND !is_numeric(
        $params[vB_dB_Query::PARAM_LIMIT]))
      return false;

    if (isset($params[vB_dB_Query::PARAM_LIMITSTART]) AND !
        is_numeric($params[vB_dB_Query::PARAM_LIMITSTART]))
      return false;

    return true;
  }
  ...
}
```

Listing 4. vBForum_dB_MYSQL_QueryDefs::getSubscriptionUsersLog

Ainsi, avec du simple tainting, sans considérer le retour du premier appel, il est fort probable d'arriver à un nombre élevé de faux positifs.

Un deuxième problème vient de la façon dont les requêtes SQL sont construites : pour sécuriser une chaine de caractères avant de l'insérer dans une requête, il faut échapper la chaine (remplacer les guillemets avec un caractère d'échappement), puis enclaver la chaine de guillemets. vBulletin utilise `$db->escape_string()` ou une méthode dérivée, puis concatène la valeur dans la requête, en prenant soin de l'entourer de quotes.

```
$where[] = 'user.username LIKE "' . $db->escape_string_like($params[
    'startswith']) . '%"';
```

Listing 5. Exemple de paramètre correctement sécurisé

Dans l'éventualité où les guillemets sont oubliés, la méthode `escape_string` est inutile.

Savoir si une variable est contrôlée ou non n'est plus suffisant : il faut savoir son type, quelles sont les contraintes auxquelles elle doit répondre, et dans quelle mesure elle a été échappée. Nous devons donc avoir recours à de **l'exécution symbolique** afin de vérifier si les contraintes imposées par `check_only` ne changent pas l'état de contrôle des variables, et si leur vérification et sécurisation suivantes permettent bien d'atteindre les requêtes SQL.

Le moteur d'exécution symbolique construit se base lui aussi sur PHP-Parser [3] afin d'extraire l'AST des méthodes. Il maintient, pour chaque variable, grâce à z3 [5], des contraintes de type (`array`, `string`, `int`, `bool`, `null`) et de valeur. De nombreuses conditions sont basées sur le fait qu'une variable soit vide ou non, et on garde donc cette information aussi. Les valeurs statiques (nombres, chaines, tableaux) sont elles aussi gardées en mémoire, afin de pouvoir au mieux simuler l'exécution du code. Chaque branche `if-else`, `foreach`, ou `switch` est évaluée si elle correspond aux contraintes, ou ignorée sinon. Si on ne peut pas déterminer si une condition est vraie ou fausse, alors elle est évaluée dans un nouveau contexte.

```
[METHOD] vB_dB_MYSQL_QueryDefs::userReferrals [Var(0, type=t_dict,
    controlled=True, key_type=None, val_type=None), Value({}), Value
    (True)]
[CODE_IF_TRUE] (6828:6828) if ($check_only)
[CODE_RETURN] (6830:6830) return !empty($params['referrerid']);
[METHOD] vB_dB_MYSQL_QueryDefs::userReferrals [Var(0, type=t_dict,
    controlled=True, key_type=None, val_type=None), Value({}), Value
    (False)]
[CODE_IF_FALSE] (6828:6828) if ($check_only)
[ASSIGN] (6834:6834) $where = "WHERE referrerid = " . $params['
    referrerid'] . " AND usergroupid NOT IN (3, 4)";    ->    Value(
    'WHERE referrerid = <REAL:CONTROLLED:?> AND usergroupid NOT IN
    (3, 4)')
[CODE_IF_DUNNO] (6835:6835) if (!empty($params['startdate']))
[CODE_IF_DUNNO] (6840:6840) if (!empty($params['enddate']))
[CODE_IF_TRUE] (6845:6845) if (!empty($params['enddate']))
[ASSIGN] (6848:6852) $sql = "SELECT username, ...";    ->    Value('
    SELECT username, posts, userid, joindate, lastvisit, email FROM
    prefix_user WHERE referrerid = <REAL:CONTROLLED:?> AND
    usergroupid NOT IN (3, 4) AND joindate >= <REAL:CONTROLLED:?>
    AND joindate <= <REAL:CONTROLLED:?> ORDER BY joindate DESC')
[ASSIGN] (6854:6854) $resultclass = 'vB_dB_' . $this->db_type . '
    _result';    ->    Value('vB_dB_MySQL_result')
```

```
[FOUND] SELECT username, posts, userid, joindate, lastvisit, email
    FROM prefix_user WHERE referrerid = <REAL:CONTROLLED:?> AND
    usergroupid NOT IN (3, 4) AND joindate >= <REAL:CONTROLLED:?>
    AND joindate <= <REAL:CONTROLLED:?> ORDER BY joindate DESC
...
[EXCEPTIONS] ------------------------------------------
[RESULTS] ---------------------------------------------
[INJ] SELECT username, ... FROM prefix_user WHERE referrerid = <REAL
    :CONTROLLED:?> AND usergroupid NOT IN (3, 4) AND joindate >= <
    REAL:CONTROLLED:?> AND joindate <= <REAL:CONTROLLED:?> ORDER BY
    joindate DESC
...
[INJ] SELECT username, ... FROM prefix_user WHERE referrerid = <REAL
    :CONTROLLED:?> AND usergroupid NOT IN (3, 4) ORDER BY joindate
    DESC
```

Listing 6. Exécution symbolique sur vB_dB_MYSQL_QueryDefs::userReferrals

Dans l'exemple 6, qui suit une exécution sur la méthode userReferrals (exemple 2) on peut voir que le code est tout d'abord exécuté avec `check_only` à `True` (ligne 1-3), puis à `False` (ligne 4). Les injections sur les trois paramètres sont bien détectées.

Il ne nous reste plus qu'à combiner les résultats des deux moteurs.

5 Résultats

5.1 Injection SQL

Après exécution du moteur de tainting sur toutes les méthodes d'API, on dénombre **500 chemins différents** pour atteindre un QueryDef par l'API, dont **345 contenant des paramètres envoyés par l'utilisateur**. Au total, **245 QueryDefs différentes** sont appelées. L'exécution symbolique sur ces résultats donne deux résultats valides.

Le premier est malheureusement accessible uniquement à des administrateurs, puisque les droits de l'utilisateur sont vérifiés dans la méthode d'API appelante. Le second est bel et bien pré-authentification :

```
Chaine: vB_Api_Content::getIndexableContent ->
        vB_Library_Content::fillContentTableData ->
        vBForum_dB_MYSQL_QueryDefs::getContentTablesData
Parametre: [nodeid]
Requete: SELECT col1, col2 FROM prefix_<IDENTIFIER> AS <IDENTIFIER>
    WHERE <IDENTIFIER>.nodeid = <REAL:CONTROLLED:?>
```

Listing 7. Injection SQL getContentTablesData

Une requête HTTP (figure 2) permet de confirmer l'injection.

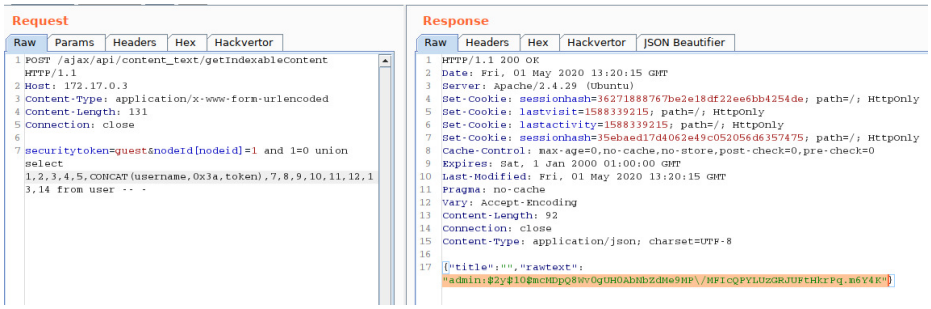

Fig. 2. Injection SQL pré-authentification

5.2 Escalade de privilège et exécution de code

L'injection SQL permet seulement de lire la base de données. Afin d'élever nos privilèges, il y a les options classiques : on peut casser le mot de passe d'un administrateur, utiliser une session existante, ou utiliser la fonction de réinitialisation de mot de passe et voler le jeton généré. En tant qu'administrateur, les possibilités pour exécuter du code sont nombreuses. On peut, par exemple, ajouter un nouveau module complémentaire.

5.3 Résultats complémentaires et suite

En lançant le moteur d'exécution symbolique sur toutes les méthodes QueryDefs, et en considérant que les paramètres d'entrées sont complétement contrôlés, on obtient 12 méthodes réellement vulnérables, et 9 faux positifs.

Toutes les informations ont été remontées à vBulletin, et la correction des vulnérabilités est en cours. L'outil sera publié sur le GitHub du groupe [2].

Comme l'API ne constitue pas toutes les façons d'interagir avec l'utilisateur, je compte adapter l'outil afin de tester aussi les autres pages du produit.

6 Conclusion

Afin de venir à bout de l'immensité du code de vBulletin, j'ai construit deux outils basiques. À eux deux, ils ont produit un résultat valide : une injection SQL pré-authentification. L'exploitation de celle-ci permet finalement d'exécuter du code sur le serveur ; on a donc une faille d'exécution de code sur la dernière version de vBulletin. Les deux outils pourront

sans doute, permettre d'auditer d'autres parties de vBulletin, et surtout, d'autres produits en PHP.

Références

1. Arbre de la syntaxe abstraite, AST. `https://fr.wikipedia.org/wiki/Arbre_de_la_syntaxe_abstraite`.
2. GitHub de Ambionics, Lexfo. `https://github.com/ambionics/`.
3. PHP-Parser, par Nikita Popov. `https://github.com/nikic/PHP-Parser`.
4. Tweet du CEO de Zerodium au sujet de CVE-2019-16759. `https://twitter.com/cBekrar/status/1176803541047861249`.
5. z3 Solver. `https://github.com/Z3Prover/z3`.
6. National Vulnerability Database. CVE-2019-16759 : vBulletin Remote Code Execution. `https://nvd.nist.gov/vuln/detail/CVE-2019-16759`.
7. Common Vulnerabilities and Exposures website. Listes de CVEs liés aux injections SQL sur vBulletin. `https://www.cvedetails.com/vulnerability-list/vendor_id-8142/opsqli-1/Vbulletin.html`.