Le 31/08/2022,

Objet : Protection contre les injections SQL

Bonjour,

Ce guide est à destination de développeurs gravitant sur nos infrastructures donc des profils 100% techniques.

Les fonctions pSQL / bqSQL et esc_sql (PS / WP) ne servent à rien si elles sont mal utilisées

Il fait suite à la vague d'attaque de S2 2022 qui a mis en exergue de mauvaises pratiques de la part des développeurs ainsi que des incompréhensions de la part de certains développeurs sur le scope de certaines fonctions natives de protection anti injection SQL des CMS.

Aux rares profils susceptibles de prendre à la légère le propos , nous vous invitons à lire ceci : https://www.touchweb.fr/aide/protection - ainsi qu'à prendre conscience que vous pouvez vous retrouver juridiquement responsabilisé au regard du RGPD : **au titre d'une malveillance par négligence volontaire** en tant que sous traitant du responsable de traitement ayant engendré une violation potentielle de l'article 32 du RGPD.

Randomiser votre préfixe de table ne vous protège pas

1. Qu'est ce qu'une injection SQL ?

C'est un type d'attaque qui permet de prendre le contrôle de votre base de données et d'y effectuer des CRUD malveillants ou des extractions malveillantes.

2. Quel est son but ?

Ils sont aussi variés qu'il y a de cerveaux torturés pour les créer, usuellement c'est assez majoritairement pour du vol de données, mais depuis 2022 via un détournement de mécaniques natives à certains CMS (comme un moteur de template susceptible d'exploiter un cache en base), cela a également été utilisé pour mettre en place des moyens de paiement frauduleux en substitution des existants (web-skimmers) - **vol de cartes bleues.**

Nous n'aborderons pas pour la suite les détournements de SELECT dans le but d'une extraction qui ne nous intéressent pas dans ce guide vu qu'ils ne sont pas "dangereux" (très dérangeants mais pas dangereux pour l'intégrité du système d'informations) en tant que tel - comprendre "de manière directe".

3. A quoi est-ce que cela ressemble ?

A ceci : https://XXX/script.php?id=1);delete+from+0test+where+1;--

4. Quel est le point commun à la quasi-totalité des injections SQL qui CRUD (les plus dangereuses) ?

L'échappement de clauses pour fermer la requête SQL initiale dans l'objectif d'en injecter d'autres - au sens large du terme.

Il est essentiel de parfaitement comprendre cette phrase car **certains développeurs pensent innocemment que les fonctions natives des CMS vont les protéger systématiquement ce qui est loin d'être une vérité.**

5. Fonctions inutiles

Vu qu'il semblerait que certains développeurs ne l'aient pas en tête, veillez à ne pas utiliser les fonctions suivantes pour vous protéger d'injections SQL - elles n'ont aucun intérêt à ces fins :

- le cast (string)

- les fonctions urldecode / strval

- la condition if($string_a_risque <= 0) ou équivalent - **Attention avec le transtypage dynamique : "1;test" > 0**

- toute autre chose qui n'est pas une fonction prévue à la protection d'une injection SQL dans sa définition officielle

On en rajoutera si on voit d'autres choses inutiles du genre - ces quatre-là reviennent trop souvent.

6. Comment vous en prémunir ?

Toutes données reçues par un inconnu tel qu'un visiteur sur votre site doit faire l'objet d'une attention particulière.

Exemple 1 - DANGER :

```
$test = $_GET['test42']

$retour = Db::getInstance()->getValue("SELECT id FROM `"._DB_PREFIX_."0test` WHERE id= ".$test;
```

Ici on constate que la valeur soumise par un guest n'est pas du tout protégée au sein de la requête SQL en soit - vous n'avez même pas besoin d'échapper quoi que se soit, vous pouvez procéder à une injection directe :

```
https://XXX/script.php?test42=1;delete+from+0test+where+1;--
```

Ce type de requête SQL est dangereuse : aucune fonction native "standard" de protection (tel que pSQL/bqSQL de Prestashop, ou esc_sql de Wordpress) ne vous protègera d'une injection SQL. Vous devez considérer comme obligatoire le fait d'encapulser vos variables dans des apostrophes / guillemets ou forcer un type spécifique au traitement de chiffres tel que (int) ou (float).

Nous insistons, les fonctions suivantes ne vous protègeront pas :
- **pour Prestashop : pSQL / bqSQL** (#1 - function pSQL / bqSQL, #2 function escape, #3 function _escape)
- **pour Wordpress : esc_sql** (#1 function esc_sql, #2 function _escape et function _real_escape, #3)

$retour = Db::getInstance()->getValue("SELECT id FROM `"._DB_PREFIX_."0test` WHERE id= ".pSQL($test);

Cette requête est toujours détournable par :

https://XXX/script.php?test42=1;delete+from+0test+where+1;-- vu qu'elle ne contient aucune apostrophe ni guillemet.

Protection: forcer un (int) - cette requête ne pourra de toute manière traitée que des chiffres :

$retour = Db::getInstance()->getValue("SELECT COUNT(id) FROM `"._DB_PREFIX_."0test` WHERE id= ".(int) $test;

Exemple 2 - DANGER :

$test = $_GET['test42']

$retour = Db::getInstance()->getValue("SELECT id FROM `"._DB_PREFIX_."0test` WHERE id IN (".$test.")";

ATTENTION avec les clauses SQL type IN(ARRAY) - les fonctions natives de la quasi-totalité des CMS ne vous protègent pas d'un détournement vu qu'une simple parenthèse suffit à échapper la clause.

Entre autres choses, les fonctions **pSQL/bqSQL pour Prestashop et esc_sql pour Wordpress ne vous protégeront pas.**

$retour = Db::getInstance()->getValue("SELECT id FROM `"._DB_PREFIX_."0test` WHERE id IN (".pSQL($test).")";

sera toujours détournable par https://XXX/script.php?test42=1);delete+from+0test+where+1;--

Deux options pour protéger ce type de requêtes SQL :

1. La très grande majorité du temps, ce type de requêtes SQL attend en entrée un tableau d'entier, vous pouvez donc rapidement le protéger ainsi :

```
$test_ori = explode(',', $_GET['test42']);
$test_ori = array_map('intval', $test_ori);
$test = implode(',', $test_ori);
```

$retour = Db::getInstance()->getValue("SELECT id FROM `"._DB_PREFIX_."0test` WHERE id IN (".$test.")";

2. Si vous souhaitez y soumettre des strings, vous devez filtrer les parenthèses fermantes (condition d'échappement de la clause)

Exemple 3 :

$test = $_GET['test42']

$retour = Db::getInstance()->getValue("SELECT id FROM `"._DB_PREFIX_."0test` WHERE name= '".$test."'";

Ici on constate que $test est encapsulé, pour la détourner, il faut donc échaper le guillemet comme ceci :

https://XXXXXX/script.php?test42=1%22;delete+from+0test+where+1;--

Vu que vous avez encapulsé la variable entre apostrophe / guillemet, les fonctions natives de protection fournies par les CMS telle que pSQL pour Prestashop répondront au besoin de protection.

Protection :

$retour = Db::getInstance()->getValue("SELECT id FROM `"._DB_PREFIX_."0test` WHERE name= '".pSQL($test)."'";

pSQL va forcer un backslash devant tous les guillemets / apostrophes fournis par l'inconnu, ce qui l'empêchera d'échapper la clause quoi qu'il se passe.

7. Est-ce que le fait d'utiliser des préfixes de tables en base non prédictibles améliore la protection ?

Des scripts kiddies oui (autrement dit des juniors), vous atténuez les risques sous réserve que le préfixe de tables que vous utilisez respecte la complexité **MINIMALE** suivante : [a-z0-9]{6} (minimum 2 milliards de combinaisons possibles).

Il est "simple" de chainer des CRUD au sein d'une injection SQL, comprendre : vous pouvez chainer des centaines de milliers de requêtes SQL pour "brute-forcer" le préfixe de tables.

Tout cela est souvent conditionné sur les stacks LAMP conventionnels à la valeur max_allowed_packet pour le SGBD (MariaDB / MySQL / Percona) et la valeur post_max_size pour PHP.

Dans nos contextes E-Commerce, il est de notoriété publique que ces valeurs sont toujours élevées (16Mo à 64Mo), dans ce cadre, il est possible de chainer plus de 500 000 requêtes SQL malveillantes PAR appel serveur, ce qui met à nu en un seul appel tout préfixe de tables dont la complexité est inférieure à [a-z0-9]{3} et en 4 appels, tous les [a-z0-9]{4}.

Cependant, par suite stabilisation d'un POC avec nos partenaires (un grand merci à Richard Stefan d'Ambris pour le POC d'origine et à Jocelyn Fournier de Softizy pour sa stabilisation dans une variante résistante à pSQL / esc_sql puis son optimisation) via un détournement de l'exemple 3 du point 10, **cela ne sert absolument à rien pour vous protéger des blackhats ayant une compétence senior en DBA.** Jusqu'à preuve du contraire, ce POC concerne tout l'écosystème PHP et plus précisément toutes les solutions exploitant MySQL ou l'un de ses forks (MariaDB / Percona).

Compte tenu de l'extrême dangerosité de ce POC, il n'est pas et ne sera jamais publié - mais a été partagé avec tous les leads dev des CMS qui nous en ont fait la demande ainsi qu'à tous ceux qui nous ont aidés et continuent de nous aider à nettoyer - activement - l'écosystème PHP.

8. Pourquoi doit-on corriger alors que l'on est protégé des formes usuelles d'injections SQL chez TOUCHWEB ?

L'informatique, comme beaucoup d'autres secteurs, souffre d'une stabilité fragile qui peut basculer à tout moment dans le chaos sur une action non maîtrisée d'un tiers.

Bien qu'il soit impossible nativement de désactiver via les surcharges de nos frontends (.htaccess) ModSecurity 2 jusqu'à preuve du contraire, comme tous les opérateurs de service et ce, pour éviter des paralysies sur les backoffices, on est contraint de lever la quasi-totalité des protections sensibles sur les adresses IP soumises aux listes blanches.

C'est un secret de polichinelle qu'un pourcentage non négligeable de clients OU intervenants techniques des clients, administrent leurs listes blanches avec une rigueur discutable.

Nous vous invitons donc à prendre en charge dans un délai raisonnable nos alertes de sécurité à ce propos pour éviter que notre CLIENT commun se retrouve exposé malgré lui au hasard d'un ajout d'une adresse IP d'un tiers n'étant en aucune manière de confiance.

9. Pourquoi former les développeurs à ce propos - souhaitez-vous faire de nous des pirates ?

Les trois dernières années ont comptées parmi les pires en matière d'incident de sécurité et la tendance s'aggrave. Il est peu probable que cette tendance s'inverse vu le contexte international.

Les pirates ont parfaitement pris la mesure de la précarité de conception d'une volumétrie importante de modules pour les CMS.

Dans ce contexte, **continuer de faire l'autruche est inacceptable, et l'innocence doit être tuée. Il est essentiel de vous phaser avec le réel - post S1 2022 : tout ce que vous ne faites pas dans les règles de l'art sera exploité de manière malveillante.**

Il faut que vous soyez en alerte quand vous êtes amené à télécharger un module sur une place de marché d'un CMS et que vous preniez le temps de réaliser un audit de sécurité, à minima sur les formes usuelles et courantes de détournement (injection SQL / injection PHP) détectables en moins de 5 minutes.

La seule et unique méthode qui permet de vous mettre en alerte, nous insistons : seule et unique, est de faire de vous des devsecops.

Autrement dit des développeurs formés aux problématiques de sécurité, qui ont conscience des menaces réelles et sérieuses de leurs marchés, qui savent les exploiter et in fine, qui sont capables de mettre en oeuvre des contre-mesures adaptées qu'ils savent stressées pour constater qu'elles répondent au besoin.

Compte tenu des us et coutumes d'au moins 50% des concepteurs de module en constaté qui partent innocemment et/ou abusivement du principe qu'il n'est pas nécessaire de protéger les codes sources dont l'accès n'est possible qu'à des administrateurs de l'application métier (souvent site E-Commerce) et ce **afin d'éviter de tous vous faire partir en dépression (réellement - ce n'est pas une emphase), on ne vous réclamera jamais une correction de vulnérabilité exploitable par un administrateur partant du principe que le simple fait que le tiers soit administrateur est une faille critique de sécurité en soit.**

Notre marché est extrêmement exposé à l'ennemi de l'intérieur (administrateur n'étant pas de confiance) et personne ne changera cet état de fait avant plusieurs années vu que cela nécessite de rééduquer au moins la moitié des développeurs de modules**.**

On vous demande uniquement de vous concentrer sur les fonctionnalités exposées et donc potentiellement exploitables par des non administrateurs (guest).

10. Pourquoi les injections fournies ne servent à rien ? Personne n'ira détruire des tables ainsi.

Par suite interaction non constructive avec des concepteurs de module qui ne comprennent pas la gravité d'un pSQL / esc_sql non encapsulé, on est contraint de fournir plus de données pour sensibiliser les tiers.

En effet, vous ne serez que très rarement confronté à ce genre de destructeurs qui n'ont aucun intérêt si ce n'est défouler les nerfs de celui qui en est à l'origine vu qu'ils sont (très) bruyants et se réparent simplement.

Pour ceux d'entre vous qui ont le niveau en développement pour comprendre que l'on peut penser que le scope d'un pSQL / esc_sql non encapsulé est limité vu qu'il n'est pas possible de CRUD "finement", ce qui est nécessaire pour, par exemple, finement altérer la table ps_configuration de Prestashop ou encore wp_options de Wordpress avec un WHERE name="KEY" qui impose l'utilisation d'un guillemet / apostrophe pour isoler une valeur précise (censé être impossible si pSQL ou esc_sql est présent vu que ces fonctions filtrent les guillemets / apostrophes).

Voici un premier poison pour les serveurs - et plus précisément les piles du pool FPM ce qui peut aggraver (fortement) l'impact d'un DDOS gravité faible à moyenne (les forts sont autogérés par la quasi-totalité des hébergeurs - ne vous en souciez pas) :

https://XXX/script.php?test42=1;select+sleep(500);--

Un deuxième, par équivalence, chiffré en hexadécimal :

https://XXX/script.php?test42=1;SELECT+0x73656c65637420736c656570283130293b
+INTO+@var_name;prepare+stmt+from+@var_name;execute stmt;

Et enfin un troisième, comme vous l'aurez compris dans le deuxième, on peut invisibiliser des requêtes SQL de complexité potentiellement forte en hexadécimal, si on pousse le vice plus loin, **on peut CRUD très finement - sans guillemet ni apostrophe** :

https://XXX/script.php?test42=1;SELECT+0x5550444154452060707 35F636F6E666967757261746 96F6E602053
45542076616C75653D27312720574845524 5206E616D653D2750535F544F4B454E5F454E41424C4527
+INTO+@var_name;prepare+stmt+from+@var_name;execute stmt;

En résumé, et sur la base de ce troisième exemple, vous devriez avoir compris si vous êtes rôdé à l'hexa qu'**un pSQL / esc_sql ne sert absolument à rien s'il n'est pas encapsulé**, autant ne rien mettre du tout, cela a le même effet.

Le propos est volontairement velu pour éviter que des juniors ne le comprennent - si vous êtes junior (moins de 5 000 heures de développement) et que vous n'avez rien compris, la seule chose à retenir est : encapsulez vos pSQL / esc_sql avec des guillemets / apostrophes - c'est tout, sinon vous mettrez en danger des projets.

11. Top 3 des arguments fallacieux

Nous avons entendu depuis 2 mois beaucoup d'arguments fallacieux, dont certains dénotent au mieux de l'innocence et au pire de l'incompétence, nous avons décidé d'en faire un TOP 3 :

TOP 1 : Cela ne sert à rien, la plupart des hébergements ont des protections anti-injection SQL ou en héritent grâce à Cloudflare (ou équivalent)

La quasi-totalité des hébergements mutualisés ne sont pas protégés contre toutes les formes d'injections SQL.

Les formes les plus dangereuses (via hexa) nécessitent un PL contraignant et beaucoup de tiers les boudent.

Par exemple, **nativement Cloudflare tout comme ModSecurity ne vous protègent de certaines formes d'injection SQL** que sous réserve que vous activiez le **très** contraignant PL4 (comprendre Paranoïa Level 4 des protections OWASP).

Si vous souhaitez le vérifier, vous avez juste à reprendre les exemples fournis (dont l'exemple 3 du point 10), l'adapter sur le lien de la page d'accueil de votre site puis constater si la page d'accueil s'affiche - si elle s'affiche : vous êtes vulnérable à ces injections SQL.

TOP 2 : Les tiers n'ont qu'à prendre des hébergements professionnels sous infogérance, ce n'est pas notre problème

Cette affirmation fallacieuse sous-entend que s'il y avait un problème majeur (Juillet 2022 ayant disparu du calendrier de 2022 chez la quasi-totalité des professionnels de l'IT), les tiers comprendraient que c'est lié à un hébergement non sécurisé.

Beaucoup de gens souffrent d'un biais que nous aimons à qualifier de "Dijkstra" contraints conjoncturellement par un monde qui veut toujours aller plus vite. Ce biais cognitif engendre des troubles du discernement, beaucoup ne liront pas que c'était un problème lié à l'hébergement, la plupart liront que c'était un problème lié à Prestashop / Wordpress et donc que c'était Prestashop / Wordpress le problème.

Vous subirez tout comme nous tous cette dégradation d'image de marque et de réputation d'écosystème, ce qui aura nécessairement un impact sur vos business.

Les problèmes de votre écosystème deviennent toujours tôt ou tard vos problèmes.

Si vous souhaitez un exemple qui nous a tous marqué à vie à l'administration système : après le drame de Strasbourg de Mars 2021, nous pensions que les gens retiendraient majoritairement que le problème était la résilience des infrastructures. Que tout le monde aurait en tête que l'heure est à la redondance. Ce n'est pas du tout ce qu'il s'est passé. Les Dijkstratistes étant majoritaires, beaucoup ont retenu fallacieusement que le problème était OVH.

TOP 3 : Cela ne sert à rien de produire des mises à jour, le client s'en fiche (...)

Soyez assuré que tous les professionnels sérieux appliqueront les mises à jour ou à défaut, un hotfix.

Pour s'assurer que les tiers prennent la pleine mesure de la dangerosité des vulnérabilités remontées, **nous greffons systématiquement à nos alertes de sécurité, les preuves de concept permettant de pirater le SHOP.**

Cela a un double effet positif, d'une part, cela forme les développeurs à stresser leurs développements pour qu'ils soient en mesure de les rendre résistants aux menaces réelles et sérieuses de leur marché (expertise devsecops) et d'autre part, cela permet d'accélerer la prise en charge, si nous ne pouvons pas l'auto-gérer pour vous aider - 99% des vulnérabilités sont traitées par le pool secops TW.

Attention : certains écosystèmes souffrent d'une obsolescence plus ou moins grave du fait d'un process de maintien à jour jugé trop lourd (et donc trop cher) par plus de 80% des concernés. **Produire des mises à jour ne répond donc que partiellement au besoin**, si vous pouvez nous fournir des hotfix, on vous en sera reconnaissant, cela nous évite une analyse différentielle et une création de patch (ce qui est la norme sur certains écosystèmes comme Magento)

Comme d'habitude, nous enrichirons le guide au fur et à mesure des découvertes de mauvaises pratiques ou du retour des développeurs.

Nous attirons votre attention sur le fait qu'**il est vital pour nos écosystèmes (Prestashop / Magento / Wordpress) que vous nous remontiez avec soin les vulnérabilités critiques de sécurité que vous découvrez** pour que l'on enrichisse nos automates de surveillance active des applicatifs métier.

Le fait de cacher la correction d'une vulnérabilité critique de sécurité est un acte irresponsable qui met en péril votre écosystème en plus d'être une violation potentielle des articles 33 et 34 du RGPD vu qu'elle invisibilise une violation hautement probable de données personnelles.

Nous comptons tous sur vous
pour consolider la protection de nos écosystèmes.

A bientôt pour un nouveau guide

L'équipe TW