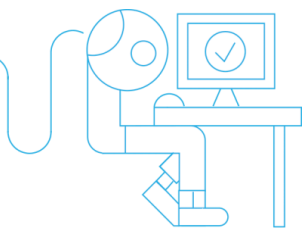
The Node.js Highway- Attacks are at Full Throttle

Contents



Introduction

Over the past few years Node.js has been gaining popularity due to its new approach to web development and its use of a very popular low entry language (JavaScript).

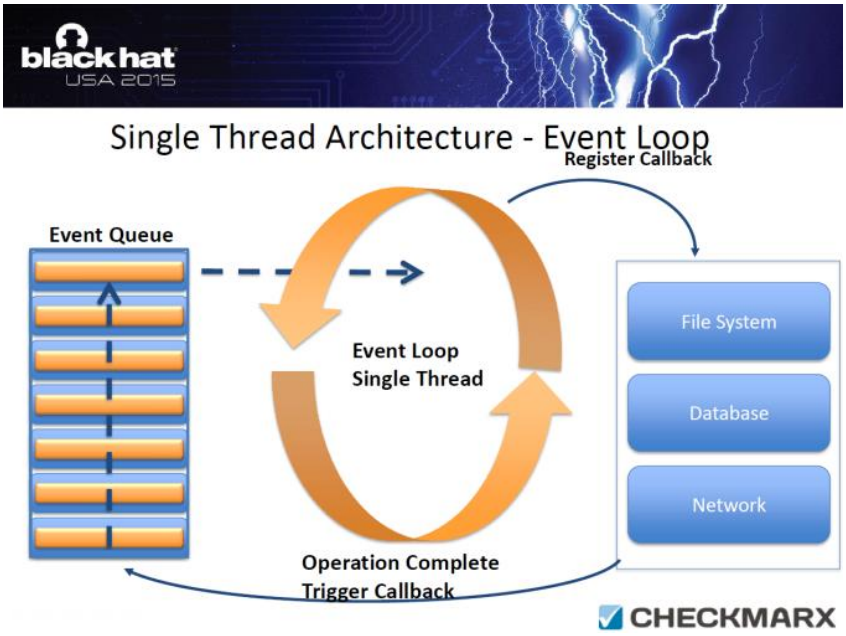
"Node.js® is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."
https://nodejs.org/

This document shares some points to look out for when developing using Node.js. We do not claim that using Node.js is not secure, but rather emphasize that developers using that platform should be aware of the potential pitfalls.

Architecture

The great thing about node.js is the fact that it is non-blocking, single threaded and event –driven. What all of this means is very clearly explained in Dan York's article at *http://code.danyork.com/2011/01/25/node-js-doctors-offices-and-fast-food-restaurants-understanding-event-driven-programming/* .

In short, when used correctly, it allows performing multiple events at a fast pace. A sample of the architecture can be seen in the following image:



However the architecture also introduces limitations that have to be taken into consideration. Some of the attack techniques mentioned in this document are directly related to wrong use of node.js which can be easily avoided by understanding the environment and architecture.

Denial of Service

Node.js is highly vulnerable to DoS. Architecturally, Node.js, being single threaded may use as much CPU as it requires to complete one task at a time. It will not switch to a different cpu-task as long as the currently running cpu-task hasn't finished. That's the reason why most of the cpu-tasks are usually short – to give other tasks the chance to get executed. When the thread is given a heavy CPU intensive calculation to perform, it will exhaust all CPU resources while blocking all other waiting events.

Example:
The following piece of code will calculate the sum of 1 through 'p' where 'p' can be any positive number. This calculation uses CPU. When 'p' is a low number, the calculation will be completed quickly and the thread will free up for the next task in the queue; however, if 'p' is a larger number, the node.js thread will take up the full CPU and hold it until the calculation is complete, even if during that time a request for the "sum" of a smaller number was received. The smaller number will have to wait until the larger one finishes the calculation. All other tasks at this stage cannot make use of the CPU and the thread is constantly busy.

```
Function sum (p)
        for (i=1;i<=p;++i)
        {
                f=f+i;
        }
```

Weak Cypto

Node.js is based on Chrome's V8 engine. V8's pseudo-random number generator (PRNG) is very popular. It is a well-known fact that V8's PRNG is weak and that its calculations are predictable. However, we have not seen a lot of exploits around that. Mostly because Chrome's implementation of the PRNG is segregated between tabs, and each one has its own seed value, so one site can't infer what would be the next values of another site.

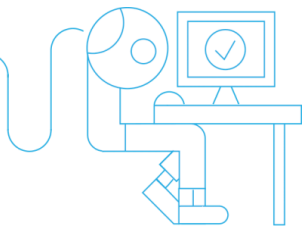
With node.js the situation is radically different. All users using the same node.js-based web-server are running within that single thread that has a single state and a single seed number. This means that given 3 consecutive "random" numbers, allows hackers to deduce what will be the next numbers, not-yet-generated.

JSON Injection

Node.js works nicely with MongoDB, being a noSql, document-oriented, json-based database. The two work together elegantly as both work with JSON natively, both for storing, retrieving and handling objects. JSON is also used by MongoDB to define the search criteria. This can lead to an interesting type of SQL-Injection (although, technically speaking, no SQL is involved, only JSON).

Let's see and example. A simple query that validates user's credentials against the data stored in MongoDB, might use the "find" method and look as follows:

```
Db.users.find({username: <username>, password: <password>});
```

(For clarity, let's assume the username and password are retrieved through the GET parameters. Obviously this will work the same for POST params)

Because node.js supports json parameters and automatically serializes and de-serializes objects - bypassing the above find request is very simple:

```
http:///server/page?user[$gt]=a&pass[$gt]=a
```

In the above case the DB will return all results including all usernames and password which are greater than 'a' (probably all).

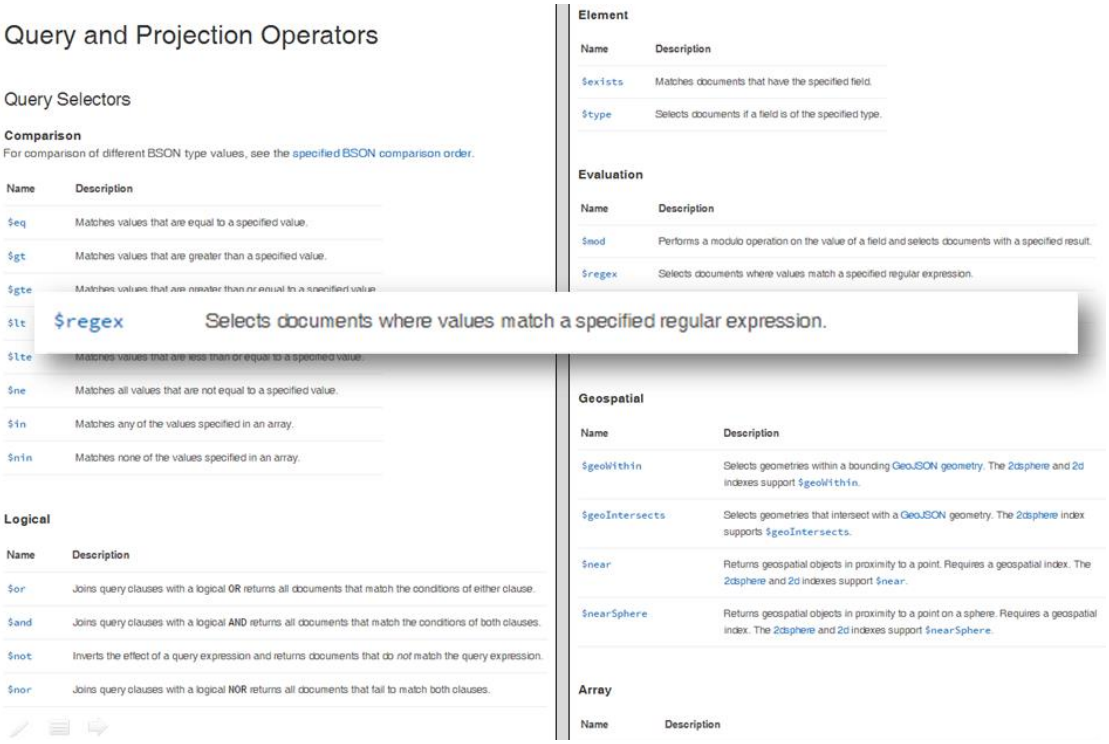
One of the common suggestions to avoid this issue, is first retrieving the hash of the password from the database for the given username, and then comparing it with the user entered password. We believe this suggestion is not bullet-proof and makes the system vulnerable to Re-DoS (Regex DoS).

```
db.users.find({username: username});

bcrypt.compare(candidatePassword, password, cb);
```

ReDoS

In the above example, we used the $GT operator to match the credentials against larger set of documents in the collection. We can also use the $REGEX operator instead.



Returning to the fact that node.js is single threaded, and that CPU intensive operations may significantly

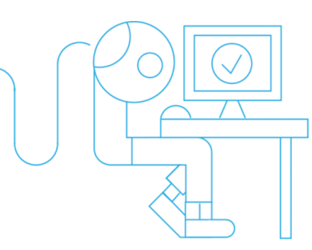
impact its functionality. Regular expressions are known to be CPU intensive so mixing these two together may easily lead to ReDoS

Looking at the example in the previous section we could easily create a request which would slow down or in some cases even completely block the node.js app and the mongoDB behind it:

```
{“username”: {“$regex”: “……..}}

Or in other words:

http://localhost:49090/?user[$regex]=^(a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a
|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a)(d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|
d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d|d)$
```



About Checkmarx:

The growing dependence on software coupled with increased exposure and usage of the Internet emphasize that software reliability is becoming increasingly critical to users. Software developers are expected to rise to the challenge and deliver applications faster than ever before which are both safe and secure. Checkmarx was founded in 2006 with the vision of providing comprehensive solutions for automated security code review. The company pioneered the concept of a query language-based solution for identifying technical and logical code vulnerabilities. The Checkmarx team is committed to both customers and technology innovation. Our research and development goes side by side with our business operations and support team to provide the best possible products and services to our customers.

See more at: https://www.checkmarx.com