Protocol Based Information Security in Distributed Systems

Turkhan Badalov, turkhan.badalov@ut.ee
Supervised by Prof. Eero Vainikko

Institute of Computer Science, University of Tartu

May 13, 2019

Abstract

Distributed systems nowadays become more like commonplace rather than special research projects. All of endpoints involved into that system use one or another form of a protocol to communicate between each other. Building security guarantees in these protocols allows to build safer and reliable systems. In this report, we consider general principles of building protocols, and perform an experiment to identify behavior of different servers as a response to SlowLoris attack.

Introduction

Information security plays a huge role in designing large, scalable and reliable distributed systems. People all over the world use distributed systems either explicitly or implicitly during their day to day works. All of these distributed systems communicate with each other to synchronize, parallelize tasks, deliver contents based on geographical location, providing/authorizing user credentials, etc. So, different types of protocols are leveraged to understand each type of requests and work in a harmony. Now, once we establish the protocol how can we make sure that our protocols are complete, not contradictory or rather miss something and not just serve for a single and simple purpose of being a "convention" to understand messages between nodes.

The Internet and modern network operates upon layered TCP/IP networking model which involves diverse protocols with different alternatives on each layer. Not checking and updates security issues in timely manner may lead to severe problems as a result of organized attacks or discovered vulnerabilities kept in secret. There are even companies that sell hot newly discovered vulnerabilities to their clients instead of submitting them to the author [7]. So, sometimes it is important to act fast before thinking over formal proofs about your system when you are under the fire of enemy requests coming from endpoints distributed over the world. Nevertheless, formal proofs add reliability and additional sense of. For this reason we talk about both aspects of protocols - theoretical parts as abstract models and practical limitations that protocols cannot usually handle.

What are communication protocols?

Protocol itself can be described as a set of rules established within some frame of activity. We need them to put some constraints on behavior of participants in order to put base for context of the activity, for example to let them know what to do or what to expect from others. More specific example could be that playing in a game involves agreeing upon rules of the game.

Another type of protocols is communication protocols. These protocols are used to establish rules between endpoints on how to process incoming mes-
sages, in other words to how understand each other. Some popular examples of communication protocols are HTTP, FTP, SMTP and others. Each of them serve for different purposes.

**Main principles of designing a protocol**

When designing a protocol, one needs to distinguish main aspects of the usage of the protocol. Is it intended to verify an identity of the person, is it used for authorization, communication, etc.?

First of all, we have to think about domain where the protocol is going to be used. Based on this information we can divide protocols into two categories:

1. **Text protocols.** Such protocols are human readable, hence easy to debug. But they make messages unjustifiably large, hard to parse. Examples are mostly all popular application layer protocols.

2. **Binary protocols.** In this case, message is represented as a sequence of binary digits (0s and 1s) which keeps messages small and compact, well structured and easy to parse. The disadvantage is that it endpoints have to agree upon representation of the message, i.e. whether it is big-endian or little-endian based message.

If it is implemented and used mostly on small devices where resources are very limited and communication happens between parts of the device, then one should prefer binary protocols due to their small size and less probability of being exposed during communications.

When it comes to designing a protocol for large audience, i.e. allowing independent endpoints communicate with each other we prefer text-based protocols because they don’t require any conventions on representation. Rather it is matter of reading and parsing the data.

Another aspect in designing a protocol is understanding states and transition between them. So, for example, the figure 1 is the state diagram taken from SMTP specification. This allows to analyze protocol as a state machine, e.g. finite automata, which makes several optimization possible like finding and removing redundant, equivalent states, in order to minimize the protocol.

![State diagram from SMTP specification](image)

**Building an abstract model of a protocol and formally proving its properties requires additional effort from a designer to justify its reliability.** Several techniques are quite common for this purpose which are unified under umbrella term known as Process Algebra. The main approaches to concurrency are:

- Calculus of Communicating Systems (CCS)
- Communicating Sequential Processes (CSP)
- Algebra of Communicating Processes (ACP)

### Distributed Denial of Service

Let’s imagine we have a well-written protocol which properties are proved formally using mathematical logic. Then it will mean that there is really no means to find a logical vulnerability in a protocol and leverage it. Does it mean we are protected? The answer is of course no because reality involves other physical limitations that attackers usually use. The simplest example might be memory of machines and other resource. Memory can be considered the most vulnerable among all of them in that sense that almost any
operation allocates some buffer in memory. So, if an attacker finds a way to force a program not to clean (release) buffers after itself or to sends so many requests that for a program to handle these requests at once there is not sufficient memory which leads to denial of service (DoS). Most of DoS attacks are based on this simple fact that resources are limited and to increase the load of a machine usually several endpoints are used to send requests simultaneously. Usually, these endpoints are located geographically in different places. That is why attacks involving several endpoints are said to be distributed, therefore distributed denial of service (DDoS).

This type of attack has been one of the most popular and even Cisco predicted that by 2022 total number of DDoS attacks will double to 14.5 millions and the number of DDoS attacks exceeding 1 gigabit per second of traffic will rise to 3.1 million by 2021, a 2.5-fold increase from 2016 [6] [4].

**Slow Loris**

Slow Loris is a very unique example of how a completely valid request may turn into a potential attack which requires less effort than usual DDoS attacks [2]. To initiate this attack what one needs to do is to initiate an HTTP request to a server and send any data with one caveat - artificially make the request slow, so that it sends one byte per second or even slower. This request forces the server to keep the connection open for very long time using more and more of its resources and releasing less and less of them causing denial of service in the end, when the server can’t afford keeping a lot of slow connections open, due to lack of memory for example.

Slowloris has several hidden features built into it due to its nature. Usually servers tend to log all requests, errors and other events in log files related to the host where the request comes. In servers with virtual hosts logs might be separated per each host. The most interesting is that when using Slowloris, symptoms of the overloaded server will not be visible in logs because servers tend to write logs after the request is completely transferred and parsed. As Slowloris transfers data with slow speed, the server doesn’t have a chance to log until the request is finished, so it waits till the end keeping connections open and eating resources silently. Of course, at the end of attack hundreds of lines might appear in log files but one can turn them into usual log files just by sending completely valid GET requests which also increases chances of staying invisible.

This is not a TCP DoS, because it is actually making a full TCP connection, not a partial one, however it is making partial HTTP requests. It’s the equivalent of a SYN flood but over HTTP. One example of the difference is that if there are two web-servers running on the same machine one server can be DoSed without affecting the other web server instance. Slowloris would also theoretically work over other protocols like UDP, if the program was modified slightly and the web server supported it. Slowloris is also not a GET request flooder. Slowloris requires only a few hundred requests at long term and regular intervals, as opposed to tens of thousands on an ongoing basis. [2].

**Modern light web frameworks against Slowloris**

During past 5 years not small amount of new java web server libraries appeared as open source projects. Simple search on github already gives us more than 2K repositories from which I found 14 libraries just on the first 10 pages of the result. For this reason, it became interesting to find out how different servers behave against slowloris attack, how long does it take to reach an ”equilibrium” when the server can’t respond anymore.

**Experiment setup**

For this experiment, we take 3 of the most popular java web frameworks appeared less than 5 years ago with an exception of NanoHTTPd which I decided to include in the list as I used it in my own projects. The following is a list of those libraries we are going to experiment with:

1. Pippo - Micro Java Web Framework [1]. It al-
allows to write the code in different styles, both in "routes" like in Ruby and annotation-based controller approach like in Spring.

2. NanoHTTPd - light-weight HTTP server designed for embedding in other applications [5]. This server is particularly interesting as it creates a new thread per each socket connection which theoretically implies infinite growth bounded by machine resources. This framework is so small that it doesn’t have any means to define routes, the only thing you have is a callback method when the request comes containing only to input arguments - Request, Response. This behavior also resembles Java EE Servlet.

3. Takes - a true object-oriented and immutable Java8 web development framework [3]. As the author states, this project is an example of a real OOP paradigm that doesn’t use static methods meaning everything is instantiated first, no null values and more.

We are going to run a very basic "Hello, world" example with each framework and perform slowloris attack on the port which the server is listening to. First, we start from 1000 connections and increase gradually by 1000 connections whenever the server repeats its behavior and doesn’t show any signs of changes. We stop the attack when the server cannot accept any more connections. Meanwhile we measure total amount of connections needed to bring the server to unresponsive state and, of course, technical details which include the amount of used memory, number of threads, type of objects created, CPU load, number of classes loaded in the memory. in addition, we print in the console whenever the request comes by using framework’s API to see if it is able to log non-complete requests coming from slowloris. To perform slowloris, we use python implementation of it from the library slowloris [2] which is executed by simply running: slowloris localhost -p port -s 1000. This attack will use 15 seconds as internal timeout between repeated requests from a socket. We can change it by changing source code.

**Pippo**
Pippo is a tiny server that has a lot of features usually provided only by scaled large frameworks. Listing 1 shows initial usage of memory when running Pippo for the first time with no incoming requests.

```java
// send 'Hello World' as response
GET("/", routeContext -> {
    System.out.println("New request");
    routeContext.send("Hello World");
});
```

Listing 1: Simple response used in the experiment for Pippo

![Figure 2: Initial memory usage in Pippo](image)

Also, it starts with 20 threads which is unusual for light servers (Figure 3).

As the first thing, we start slowloris process with 1000 requests. We notice that actually Pippo limits total number of requests by 130 though it proceeds with serving clients. So, we increased number of slowloris instances and ended up with complete denial of service, where the memory usage reached "equilibrium" state, where the usage takes the form of a pattern.

The interesting thing to note is that the attack left no logs after its execution neither during running, nor after stopping the attack, making this attack perfect directed to Pippo based applications.
We start with simple "Hello, world" example shown in the listing 2.

```java
public class App extends NanoHTTPD {
    @Override
    public Response serve(IHTTPSession session) {
        System.out.println("New connection!");
        String msg = "Hello, server";
        return newFixedLengthResponse(msg);
    }
}
```

Listing 2: Simple response used in the experiment for NanoHTTPd

The following is the initial state of the server: 1. Average memory consumption: 25MB 2. Number of loaded classes: 1608. 3. CPU usage: approximately 0.0% 4. Number of threads: 12. Where one of them is the server listener thread named "NanoHTTPd Main Listener".

We start from one slowloris process with 1000 connections and timeout of 15 seconds. After running the attack for some amount of time, we observe the general pattern shown in the figure 6. We can see repetitive sparks of 1000 threads with approximately 30 seconds gap between them. This is explained by the default timeout of 5 seconds for reading from a socket. Whenever slowloris sends the first batch of messages for each socket, all threads are created while NanoHTTPd waits for incoming messages. As the timeout in slowloris is 15 seconds, all those threads are stopped because of timeout exception in NanoHTTPd but it is not visible in slowloris yet because of idling. When the time for the next wave arrives, slowloris due to TCP implementation is operating systems still thinks it sends the data while it is actually only sent to kernel buffer and only in the third wave, i.e. after 30 seconds, it realizes that the connection is dropped and recreates a new one instead which causes the creation of a new thread. In that sense, we actually can’t leverage the all essence of slowloris attack because our connections are just kept closing.

Even after 39 instances of slowloris process with...
1000 connections per each process, server was performing quite well. As you might have guessed, because of timing issues, we never reach 39000 connections open at the same time (Figure 7). You can notice how after the read line the pattern doubled. This happened because we launch two instances at that moment and later we launch sequentially up to 39 instances.

![Figure 7: 39 instances of slowloris attacking the server](image)

As a side effect when the attack is stopped, the server starts throwing infinitely many exceptions and keeps calling callback ”serve” method. Most probably it is a bug in the code of the server, as the number of calls to the method is more than total number of requests sent to the server. You can see the spike in memory usage which appears in the end due to this glitch in the figure 8.

![Figure 8: Memory usage of the server during 39 instances of slowloris attacking the server](image)

As a next thing, let’s change timeout per connection to 4 seconds so that the server always gets some data before its default timeout of 5 seconds. Then, we get constant number of threads opened at the same time (Figure 9). Also, NanoHTTPd has a limit on header size and it reads body content of a request only when we call session.parseBody() inside of serve() method. This increases its immunity against slowloris in two ways:

- Regardless of how much data slowloris sends, the server just reads fixed amount of it as a header and leaves the rest for later time. If the server never uses request body, then the server uses constant amount of memory (Figure 9 the top graph). This makes slowloris useless mean of attack.

- Even if the server reads the request body, most probably it doesn’t do it for all routes. Thus, the attacker has to spend some time finding a proper route to perform an attack. Additionally, NanoHTTPd saves the large request body to a temporary file. Taking into account that storage is cheaper than RAM and modern devices have high read/write speed, it makes slowloris meaningless. Because in this case, the goal might be to either overflow storage size or reach maximum limit of opened files where none of these limitations depends on the speed. Vice versa, the more requests and the faster we send, the sooner we can reach these limitations.

![Figure 9: Constant memory usage and constant number of threads while single instance of slowloris attacks the server](image)

Finally, let’s limit JVM maximum heap size to 40MBs. We reach that limit after running third in-
stance of slowloris (meaning slightly more than 2000 connections). As a consequence we get series of errors (Figure 11) about lack of memory and finally the program exits as the main thread also stops working.

Figure 10: Series of errors after reaching heap size limitation

Figure 11: Final state of the server with memory usage at the top and number of threads at the bottom with 40MBs of maximum heap size

Takes

This server looked very interesting from the beginning as it actually limits the number of running connected clients (threads) by only 1 thread allowed at most. Executing one single instance of attack was sufficient to block service of the server.

Also, once attack is stopped a single log appears in the end but in the middle of attacking.

Conclusion

We saw main aspects in designing of a protocol and what are possible limitations of protocols. Specifically, we reviewed that there are text-based and binary protocols, their scope of usage. Protocol which contains states can be described in terms of finite automata and use all finite automata techniques and algorithms. Finally, we considered some examples of DoS attacks with Slowloris being remarkably interesting among them.

We ran an experiment with 3 popular lightweight web frameworks and tested slowloris on them. As a result, the most resistible was NanoHTTPd staying alive and serving every request by using all available resources. Pippo used a good technique of limiting total number of simultaneous requests being served. This makes the server vulnerable against slowloris because long and multiple connections reserve connections from the pool of connections preventing others from connecting to the server. Takes was the easiest one to attack. It limited number of parallel connections to one. Either it is done intentionally or it is configurable. Nevertheless, by default it refused to serve further incoming requests.

References

https://www.a10networks.com/resources/articles/5-most-famous-ddos-attacks


