High Layer DDoS Attacks - Slowloris

BENEDIKT THOMA, Technische Universiät München, Deutschland

1 KURZZUSAMMENFASSUNG

Diese Arbeit gibt einen Überblick darüber, wie der DoS/DDoS Angriff "Slowloris" funktioniert, wie er implementiert werden kann, welche Auswirkungen er auf gängige Webserver hat und wie man ihn mitigieren kann. Dabei werden alle, für das Verständnis der Arbeit, nötigen Grundlagen erklärt.

2 EINLEITUNG

In 2018 wurde für die Datenmenge, die eine Distributed Denial of Service (DDoS) Attacke pro Sekunde erreicht hat, ein neuer Weltrekord aufgestellt. Dabei wurde der Dienst "Github" durch die Attacke mit einem Volumen von 1,35 Tb/s angegriffen und war für einige Minuten gar nicht oder nur eingeschränkt erreichbar. Außerdem wurde Anfang 2018 noch ein weiterer Angriff mit über 1 Tb/s aufgezeichnet [1].
Dies sind Rekordzahlen und spiegeln klar den Trend zu immer größeren und immer häufigeren Denial of Service Attacken wieder. Das ist vorallem deswegen kritisch, weil die meisten Unternehmen nur Infrastruktur haben, welche Traffic zwischen 10 bis 50 Gb/s bewältigen kann [2]. Im Kontrast dazu erreichten in Q1 2017 ungefähr 23% aller aufgezeichneten DDoS Attacken Volumen von über 10 Gb/s [3].
Insbesondere interessant ist dabei, ob solche Attacken ausschließlich von Individuen herbeigeführt werden können, welche entweder über sogenannte Zero-Day Exploits oder Botnets verfügen, oder aber auch mit beschränkter Infrastruktur erstellt werden können. Außerdem ist dabei die Funktionsweise verschiedener Attacken interessant, welche auf den unterschiedlichen Schichten des OSI-Modells agieren. Im Detail habe ich mir dabei eine Attacke angeschaut, welche auf Layer 7 (Anwendungsschicht), auf Basis des HTTP Protokolls agiert und wie man mit dieser Attacke die zwei weit verbreitesten Webserver angreifen kann. Mein Beitrag dabei ist eine Einführung in den sogenannten Slowloris Angriff mit einer Darstellung von zwei verschiedenen Implementierungsmöglichkeiten und einer tatsächlichen Implementierung in golang.

Author's address: Benedikt Thoma, Technische Universiät München, Fakultät für Informatik, Boltzmannstraße 3, Garching Forschungszentrum, Garching bei München, Bayern, 85748, Deutschland.

XXXX-XXXX/2018/7-ART1

3 HTTP GET HEADER

Das HTTP-Protokoll ist einer der grundlegenden Bausteine des Internets. Es ermöglicht die Übertragung von Daten über Computernetze auf der Anwendungsschicht. Hauptsächlich wird es benutzt um das Laden von Websites in einem Browser zu realisieren. HTTP braucht zur Kommunikation zwischen zwei Endpunkten ein zuverlässiges Transportprotokoll, welches den tatsächlichen Austausch von Daten realisiert. In nahezu allen Fällen wird dafür TCP verwendet.
In HTTP ist ein Datenaustausch in zwei Nachrichten aufgeteilt. Die anfragende Nachricht, auch Request genannt, enthält alle notwendigen Daten, damit der Empfänger des Requests mit der so genannten Response antworten kann. Nachrichten bestehen dabei immer aus zwei Teilen, dem Header und dem Body. Im Header stehen dabei wichtige Informationen, die sich zwischen Request und Response zu großen Teilen unterscheiden.
Für Slowloris ist dabei nur der Request wichtig, da alle Antworten, die auf Reqeusts bei der Attacke gesendet werden, für die Attacke selbst nicht brauchbar sind.
HTTP-Requests setzen sich aus einigen Pflichtfeldern und vielen optionalen Feldern zusammen. Als erstes steht dabei immer die Art des Requests. Es gibt mehrere Arten, die einzig wichtige für Slowloris ist dabei die GET Anfrage. HTTP-GET wird verwendet um dem Server mitzuteilen, dass er mit der Ressource antworten soll, die durch den Pfad nach GET spezifiziert wird. Nach dem Pfad wird die verwendete HTTP Version angegeben. Danach kommt das Host Feld, das es möglich macht, unter der Server IP mehrere Websites mit verschiedenen DNS Namen zu hosten. Das Host Feld ist dabei unter HTTP 1.0 noch optional, ab 1.1 ist es allerdings verpflichtend.
Im Anschluss kommen dann optionale Felder wie Accepted Language, mit dem sich spezifizieren lässt, in welcher Sprache die Response gesendet werden soll, oder auch User-Agent, welches angibt, von welchem Client aus die Anfrage gesendet wurde. Das ist wichtig, da verschiedene Clients unter Umständen die Response-Bodys verschieden interpretieren könnten, oder aber zusätzliche Headerfields benötigen.
Unabhängig davon kann man beliebig selbst noch Felder im Header übertragen, ob diese valide sind oder nicht. Früher war es Konvention, selbst erstellte Parameter mit dem Präfix 'X-' zu versehen, inzwischen wird aber von der Praxis abgeraten, da sie in einigen Fällen Rückwärtskompatibilität nicht erlaubt.
Wichtig für Slowloris ist die Eigenschaft, dass der Header durch zwei Newlines, signalisiert duch zwei CRLF Zeichen, vom Body abgetrennt und damit beendet wird. (vgl. [4])

4 WEBSERVER

4.1 Architekturen von Webservern

In diesem Abschnitt geht es um Webserver, welche auf HTTP oder auch HTTPS Basis arbeiten. Der einzige Unterschied dabei ist, dass bei HTTPS der Server auf einem anderen Port, nämlich Port 443 anstatt Port 80, agiert und dass die Kommunikation bei HTTPS mit TLS verschlüsselt ist. Webserver, welche auf HTTP oder HTTPS basieren, brauchen eine Definition dafür, wie sie viele gleichzeitige Anfragen abarbeiten können. Dabei gibt es verschiedene Ansätze, welche im Grund auf zwei Schemata basieren.

4.1.1 Prefork/Threadbased.

Bei Prefork erstellt der Main Server Prozess, in dem die Kontrolllogik des Servers läuft, zur Startzeit sogenannte Handler. Diese Handler oder auch Worker sind dazu da, die eingehenden Anfragen anzunehmen und abzuarbeiten.

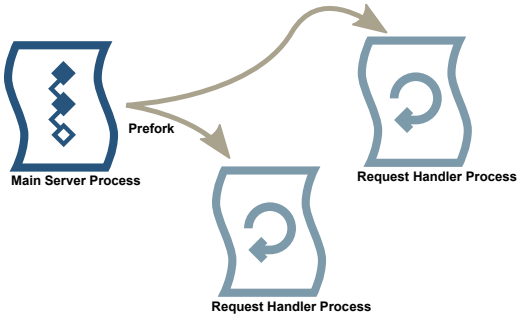

Fig. 1. Prefork Schema [5]

Diese Worker können dabei verschieden umgesetzt sein. Der Name Prefork impliziert zwar, dass die Worker durch Prozesse realisiert sind, welche durch den Systemcall fork() erstellt werden, jedoch ist die dahinterstehende Idee bei Threadbased Multiprocessing Schemata dieselbe.
Handler können durch viele Strukturen implementiert sein, welche Multiprocessing erlauben. Drei Beispiele sind dabei klassische Prozesse, Threads, oder aber auch im Fall von Golang sogenannte Routinen.
Bei Prefork sind Handler immer nur für eine Anfrage gleichzeitig zuständig. Der Handler bekommt nach dem Verbindungsaufbau über TCP die Verbindung übergeben und arbeitet dann alle Schritte alleine ab. Der Handler ist dabei exklusiv dieser einen Verbindung zugeordnet.

4.1.2 Event-driven.

Bei Event-driven Server Architekturen ist der Aufbau von Grund auf verschieden von einer Threadbased Architektur. Der Main Server Process erstellt einen oder mehrere Worker, diese sind aber nicht mehr exklusiv einer Verbindung zugeordnet.

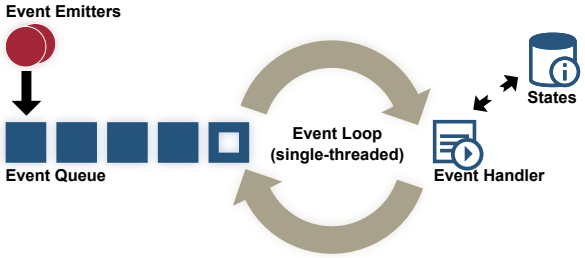

Fig. 2. Event-driven Schema [5]

Ereignisse in Verbindungen werden bei diesem Ansatz durch Events abgebildet. Diese Events werden dann in einer Warteschlange verwaltet. Aus dieser Warteschlange nimmt sich dann der Handler ein Event heraus und arbeitet alle notwendigen Schritte für dieses Event ab. Der Status einer Verbindung wird dabei in einer geeigneten Datenstruktur gespeichert.
Nach der Verarbeitung eines Events nimmt sich der Handler dann entweder ein bereits existierendes Event aus der Queue oder wartet, bis neue Events in die Warteschlange eingereiht werden. (vgl [5])

4.2 Architektur von Apache

Apache ist einer der drei am weitest verbreiteten Webserver, die es gibt. Der Apache Webserver unterstützt dabei mehrere Modi, in welchen er simultane Anfragen abarbeiten kann. Diese Modi sind dabei durch Module realisiert, welche MPM (Multi Processing Module) heißen. Dabei gibt es vorallem drei Module abseits der Betriebsystem spezifischen MPM, die sich in ihrer Art des Processing Schemas unterscheiden.
Das Modul MPM_Prefork setzt dabei das bereits beschriebene Prefork Schema mit Prozessen als Handlern und ohne Thread Unterstützung um [6].
MPM_Worker realisiert das Threadbased Schema in Kombination mit dem Prozessschema. Der Steuerprozess erstellt dabei einige Kindprozesse, welche ihrerseits dann eine bestimmte Anzahl an Serverthreads und einen Listenerthread bereitstellen, welcher Verbindungsanfragen annimmt und an die Serverthreads zu Bearbeitung weitergibt. (vgl. [7])
MPM_Event basiert auf MPM_Worker. Der Unterschied ist, dass bei MPM_Event das Verwalten der Verbindungsendpunkte (Sockets) auch in den Listenerthread des MPM_Worker ausgelagert wird. Damit verwaltet dieser alle Sockets die in einem der folgenden Zuständen sind: Listening, Keep Alive, Work done und Send Data to Client. (vgl. [8])

Apache ist damit ein Webserver, der auf einer eher klassischen Architektur basiert, in dem er das Abarbeiten von Anfragen immer exklusiv auf eine Ablaufstruktur mapped, welche wahlweise durch einen Prozess oder Thread implementiert ist.

4.3 Architektur von Nginx

Nginx ist ein Webserver, der auf einer Event-driven Architektur basiert. Er besteht aus einem Master-Prozess, der alle Operationen durchführt, welche mehr als Standardberechtigungen brauchen. Dann gibt es zwei Prozesse, die auf dem internen Cache arbeiten, welche hier nicht weiter von Interesse sind. Die restlichen Prozesse sind dann die Worker, welche die eigentliche Arbeit verrichten.
Die Worker verarbeiten dabei alle Input/Output Operationen und das Abarbeiten der eingehenden Anfragen. Dabei hat jeder Prozess nur einen Thread und die Prozesse arbeiten unabhängig voneinander. Diese Workerprozesse folgen der bereits erläuterten Event-driven Architektur, wobei jeder Workerprozess mit einer eigenen Queue ausgestattet ist und es nicht eine Queue für alle Prozesse gibt. (vgl. [9])

5 SLOWLORIS

5.1 Funktionsweise

Slowloris wurde designed, um Webserver anzugreifen. Da der Angriff auf HTTP basiert, sind Dienste, die auf anderen Protokollen basieren, folglich auch nicht angreifbar durch Slowloris.
Das Interessante an Slowloris ist, dass die Attacke wenige lokale Ressourcen und kaum Bandbreite benötigt. Daher ist Slowloris auch schon mit nur einem normalen Heimrechner oder Laptop effektiv ausführbar. Das kommt daher, dass die Idee hinter Slowloris ist, nicht durch Aufbrauchen der Bandbreite vor dem Targetsystem oder durch Überlasten des Ziels eben jenes zu Fall zu bringen, sondern alle möglichen Verbindungsslots eines Servers zu belegen und möglichst lange zu halten.
Dabei sieht der Ablauf folgendermaßen aus:

```
öffne Verbindungen, bis keine neuen Verbindungen mehr
  angenommen werden oder die konfigurierte Anzahl an
  Verbindungen erreicht ist;
sende auf allen Verbindungen initiale Header-Pakete;
while true do
    sleep(timeout);
    sende Timeoutreset Pakete;
    versuche vom Target geschlossene Verbindungen neu zu
      erstellen;
    if wenn neue Verbindungen erstellt wurden then
        sende auf neuen Verbindungen initiale
          Header-Pakete;
    end
end
```

Algorithm 1: Slowloris Logik

Slowloris etabliert also so viele Verbindungen wie möglich zum Ziel auf TCP Port 80, welcher per Konvention der Port für HTTP Verbindungen ist. Wenn der Angriff auf HTTPS erfolgen soll, müssen die Verbindungen auf TCP Port 443 geöffnet werden und der Traffic mit TLS zusätzlich verschlüsselt werden.
Dieses Etablieren von Verbindungen auf einem Server, der aktive Nutzer hat, geschieht nach und nach. Wenn Nutzer ihre Verbindungen trennen, nimmt Slowloris so lange diese freigewordenen Plätze ein, bis alle möglichen Plätze belegt sind.
Auf jeder der Verbindungen wird dann ein syntaktisch und auch semantisch korrekter HTTP GET Header an das Zielsystem geschickt. Ob die dabei angefragte Seite tatsächlich existiert ist egal, da die Anfrage nie vervollständigt wird. Anschließend an den Header wird in Intervallen, welche kleiner sind als der Timeout des Servers für zu langsame Verbindungen, ein Custom HTTP Field nachgeschickt, um den Timer wieder zu resetten. Ziel davon ist, dass die jeweilige Verbindung solange offen gehalten wird, bis sie in den größeren Timeout des Servers läuft, welcher die Zeit limitiert, um eine Anfrage fertig zu übertragen.
Bei diesem Angriff muss beachtet werden, dass ein HTTP Header durch ein doppeltes CRLF Zeichen beendet wird; daher darf diese Characterfolge nie gesendet werden. Die tatsächliche Form der gesendeten Pakete ist der Implementierung zu entnehmen [14].
Da das Auslasten des Webservers nicht stark ressourcenbelastend ist, ist ein großer Vorteil von Slowloris, dass andere auf der gleichen Infrastruktur gehostete Systeme nicht beziehungsweise kaum durch den Angriff auf den Webserver eingeschränkt werden. (vgl. [10] und [11])

5.2 Golang für Slowloris

In diesem Abschnitt möchte ich auf ein paar spezifische Charakteristika der Sprache Golang eingehen, da die Implementierung von Slowloris in Golang vorgenommen wurde und die Wahl dieser Sprache oder ähnlicher Sprachen einige Vorteile mit sich bringt, welche den Angriff auf große Systeme vereinfachen.
Das Multiprocessing System in Golang funktioniert nicht wie bei vielen klassischen Sprachen über Threads, sondern über sogenannte Routinen. Routinen lassen sich als light weight Threads beschreiben und sind damit sehr viel ressourcenärmer als Threads oder gar Prozesse. Das ermöglicht es der Sprache sehr viele Routinen gleichzeitig zu starten. Eine grobe Schätzung für die Anzahl an Routinen, die erstellt werden können, sind 150.000 Routinen pro Gigabyte an Arbeitsspeicher. Diese Schätzung ist implementierungsspezifisch [14].
Da bei der Slowloris Attacke der Kontrollfluss den größten Teil der Zeit nur schläft und wartet, bis ein neues Custom Field gesendet werden muss, ist die Belastung für sonstige Komponenten eines Computers sehr niedrig. Diese Kombination aus Routinen, welche wenig Speicher benötigen, und einer niedrigen Belastung anderer Teile des Computers ermöglicht es einer Slowloris Implementierung in Golang auch sehr große Webserver anzugreifen.

5.3 Architektur der Implementierung

Es gibt zwei Möglichkeiten die Architektur von Slowloris zu konzipieren und zu implementieren, wobei die eine Variante sehr viel potentere Systeme erfolgreich angreifen kann als die andere.

5.3.1 Single-threaded.

Die erste Möglichkeit ist eine sequentielle Implementierung ohne Multiprocessing Strukturen. Dabei werden erst alle Verbindungen aufgebaut und dann nacheinander auf allen Verbindungen die Pakete verschickt. Das hat zur Folge, dass die Zeit, die der Hauptthread schlafen muss, bevor er die neuen Resetpakete verschickt, nicht pauschal abgewartet werden kann, sondern diese bei vielen Verbindungen in Abhängigkeit der Anzahl an Connections berechnet werden muss. Außerdem, angenommen es stehen genug lokale Ressourcen zur Verfügung, dauert es bei ausreichender Anzahl an Verbindungen zu lange, um alle Resetpakete zu verschicken, sodass die erste Verbindung bereits austimed, bevor bei ihr wieder ein Resetpaket verschickt werden würde. Dieser Aufbau ist je nach Ressourcen durchaus valide, allerdings ist hier generell eine Architektur mit Multiprocessing zu bevorzugen.
Der Pseudocode eines sequentiellen Aufbaus sieht in etwa so aus:

```
Data: maxConn = konfigurierte maximale Anzahl an zu
      öffnenden Verbindungen
sockList = Liste, um alle Verbindungen zu verwalten
while numConn < maxConn do
    conn, error = createConnection(target);
    if error then
        break;
    end
    conn.Send(initPaket);
    sockList.Add(conn);
end
while True do
    foreach c in sockList do
        error = c.Send(resetPacket);
        if error then
            sockList.Remove(c)
        end
    end
    dif = (maxConn-len(sockList));
    foreach _ in range dif do
        conn, error = createConnection(target);
        if error then
            break;
        end
        conn.Send(initPaket);
        sockList.Add(conn);
    end
end
```

Algorithm 2: Slowloris Sequentieller Pseudocode (vgl. [12])

5.3.2 Multi-threaded.

Die andere Möglichkeit der Implementierung stützt sich stark auf Multiprocessing Kapazitäten. Damit werden die vorher beschriebenen Probleme einer sequentiellen Implementierung gelöst, die Attacke kann theoretisch beliebig skalieren und das resultierende Programm kann eventuell vorhandene Rechenleistung und zusätzliche Prozessorkerne besser beziehungsweise überhaupt nutzen.
Für die Multiprocessing Implementierung gibt es verschiedene Möglichkeiten, wie man diese architekturell aufbauen kann. Es wäre möglich, einfach mehrere Instanzen der sequentiellen Implementierung zu starten, dabei haben die einzelnen Instanzen aber wieder die theoretischen Probleme der sequentiellen Implementierung.
Die in der tatsächlichen Implementierung gewählte Architektur benutzt dabei den Vorteil, den Golang Routinen gegenüber klassichen Threads haben, nämlich dass man sehr sehr viele Routinen gleichzeitig starten und laufen lassen kann. Das führt zu einer Klassifizierung in zwei Arten von Routinen, wobei eine davon nicht zwangsweise nötig, aber durchaus aus Performance Gründen sinnvoll ist.
Die erste Art von Routine ist dafür da, die einzelnen Verbindungen zu dem Zielsystem zu handeln. Sie übernehmen alle Aufgaben, die in der Connection zu erledigen sind, nachdem diese einmal aufgebaut wurde.
Die zweite Art von Routinen ist zuständig, alle Routinen erster Art zu erstellen und sicher zu gehen, dass die Anzahl dieser zu jedem Zeitpunkt maximiert wird. Dabei ist es empfehlenswert, bei einer Golang Implementierung eine Routine zweiter Art pro Prozessorkern zu erstellen.
Der Pseudocode für diese Architektur sieht dann folgendermaßen aus:

```
Data: numCores = Anzahl an Processingcores
foreach _ in numCores do
    startRoutine(initHandlersFunc);
end
```

Algorithm 3: Slowloris Main Pseudocode

```
Data: maxConn = konfigurierte maximale Anzahl an zu
      öffnenden Verbindungen
counter = Zählervariable die die Anzahl an aktuell offenen
Routinen pro Core mitzählt
while True do
    while counter < maxConn do
        conn, error = openConnection(target);
        if error then
            break;
        end
        startRoutine(handleConnection, conn, *counter);
        counter++;
    end
    sleep(1s);
end
```

Algorithm 4: initHandlersFunc

```
Data: conn = Connection zum Target System für diese
      Routine
counter = Pointer zu Zählervariable
timeout = konfigurierter Timeout welcher kleiner sein muss
als der Connectiontimeout des Target Servers
conn.Send(initPacket);
while True do
    sleep(timeout);
    error = conn.send(resetPacket);
    if error then
        counter−;
        exitRoutine();
    end
end
```

Algorithm 5: handleConnection

6 AUSWIRKUNGEN VON SLOWLORIS

Für die im Folgenden beschriebenen Auswirkungen wird davon ausgegangen, dass bei den jeweiligen Webservern keine Schritte unternommen wurden, die gegen Slowloris vorgehen.

6.1 Apache

Da Apache in allen MPM eine Architektur aufweist, welche die Zielarchitektur von Slowloris ist, ist Apache höchst angreifbar. Die Dauer, bis ein Apacheserver keine verbleibenden Verbindungen mehr bereitstellen kann, ist von einigen Faktoren abhängig: Wie schnell kann Slowloris Verbindungen zum Server aufbauen ? Wie lange dauert es, bis alle oder fast alle aktuell aktiven Nutzer ihre Verbindung getrennt haben? Für wie viele gleichzeitige Verbindungen ist der Server konfiguriert?
Die tatsächlichen Auswirkungen auf Apache sind dabei immer von der Konfiguration von Apache abhängig. Außerdem besteht auch eine Abhängigkeit von Modulen, die eventuell installiert sind.

6.2 Nginx

Bei Nginx ist die Verwundbarkeit eine andere, bei weitem nicht so drastische wie bei Apache. Event-driven Architekturen sind rein konzeptionell nicht von Slowloris angreifbar. Daher wird Nginx als Event-driven Webserver normalerweise nicht als von Slowloris betroffen gelistet.
Was dabei allerdings außer Acht gelassen wird ist, dass technisch gesehen Nginx durchaus von Slowloris angegriffen werden kann. Dies passiert nicht mehr dadurch, dass alle Worker blockiert werden, sondern dadurch, dass die maximale Anzahl an gleichzeitig offenen Filedeskriptoren ausgelastet werden kann. Diese liegt in der Defaultkonfiguration von Nginx bei lediglich 1024, welche von Slowloris einfach alle belegt werden können. Da Nginx daraufhin keine neuen Verbindungen mehr annehmen kann, ist der Server dann lahmgelegt.
Die maximale Anzahl an Filedeskriptoren kann zwar stark erhöht werden, aber da die Golang Implementierung auch viele Hundertausend bis mehrere Millionen an Verbindungen öffnen kann, ist das nicht wirklich eine Lösung.

6.3 Mitigieren von Slowloris

Es gibt einige sehr einfache und gleichzeitig effiziente Wege, Slowloris zu mitigieren.
Die dabei effizienteste Methode ist die sogenannte Packet Inspection. Dabei untersucht ein Computer oder Server alle Pakete, die durch ihn hindurch übertragen werden, genau auf ihren Inhalt und ihre Struktur. Das ermöglicht es, unvollständige Pakete bei einem anderen Server zwischen zu lagern und nur nach Vervollständigung an den Webserver weiterzuleiten oder diese sofort zu verwerfen. Das führt dazu, dass der Webserver von dem Angriff nicht erreicht wird.
Eine weitere Möglichkeit ist das Limitieren der Anzahl gleichzeitiger Verbindungen zu dem Server pro IP. Da für Slowloris eine valide TCP Verbindung nötig ist, kann die anfragende IP in diesem Fall auch nicht gespooft, also manipuliert werden. Mit dieser Maßnahme wird die größte Stärke von Slowloris entfernt, dass die Attacke von einem einzigen Computer ausgeführt werden kann.
Eine andere Option ist es, die Timeouts des Webservers anzupassen. Dies ist aber nur eine scheinbare Lösung, da sich lediglich der Aufwand für einen Angriff erhöht, ihn aber nicht verhindert. (vgl. [10])
Für Apache gibt es auch noch weitere Möglichkeiten in der Form von speziellen Modulen, die entwickelt wurden, um den Effekt von Slowloris zu verhindern. Ein Beispiel ist mod_antiloris. Diese Module verlassen sich aber auch nur auf die oben genannten Arten der Mitigierung, im Fall von mod_antiloris auf das Limitieren der gleichzeitig möglichen Verbindungen pro IP. (vgl. [13])

7 FAZIT

Slowloris ist eine Attacke, die noch viel zu oft durchgeführt werden kann, da einige sehr simple Schritte nicht unternommen werden, um einen Angriff durch sie zu verhindern. Es bedarf meistens nur einigen kleinen Zusatzschritten bei der Installation eines Webservers, um diesen permanent zu schützen, insofern sollten diese Arbeitsschritte immer unternommen werden. Vorallem wegen der Menge an frei erhältlichen Tools, welche es auch einem Nutzer ohne technischem Know-How erlauben, erfolgreiche Angriffe durchzuführen, ist diese Attacke sehr gefährlich.

REFERENCES

[1] Alexander Khalimonenko, Oleg Kupreev, Ekaterina Badovskaya,
DDoS attacks in Q1 2018.
https://securelist.com/ddos-report-in-q1-2018/85373/
26.04.2018.

[2] Robert Lemos,
How DDoS Attacks Techniques Have Evolved Over Past 20 Years.
http://www.eweek.com/security/how-ddos-attacks-techniques-have-evolved-over-past-20-years
16.03.2018

[3] Jeff Goldman,
Average DDoS Attack Size Surged in Q1 2017.
https://www.esecurityplanet.com/network-security/average-ddos-attack-size-surged-in-q1-2017.html
25.05.2017

[4] IETF (Internet Engineering Task Force),
RFC 1945, 2616, 7540, 7541, 7230-7235.
https://tools.ietf.org/html/

[5] Benjamin Erb,
Concurrent Programming for Scalable Web Architectures.
http://berb.github.io/diploma-thesis/original/042_serverarch.html

[6] *MPM_Prefork*
https://httpd.apache.org/docs/2.4/de/mod/prefork.html

[7] *MPM_Worker*
https://httpd.apache.org/docs/2.4/de/mod/worker.html

[8] *MPM_Event*
https://httpd.apache.org/docs/2.4/de/mod/event.html

[9] Owen Garrett,
Inside NGINX: How We Designed for Performance & Scale.
https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/
10.06.2015

[10] SlashRoot,
SLOWLORIS: HTTP DOS(Denial Of Service)attack and prevention.
https://www.hackersclub.io/single-post/2016/03/04/SLOWLORIS-HTTP-DOSDenial-Of-Serviceattack-and-prevention
04.03.2016

[11] *SLOWLORIS.*
https://www.incapsula.com/ddos/attack-glossary/slowloris.html

[12] *slowloris.py*
https://github.com/gkbrk/slowloris/blob/master/slowloris.py

[13] *mod_antiloris*
https://coderwall.com/p/hmgy3q/mod_antiloris-anti-slowloris-apache-module

[14] *Implementierung von Slowloris*
https://github.com/grekhor/slowloris