Exploiting the Congestion Control Behavior of the Transmission Control Protocol

Stefanos Harhalakis
Department of Informatics
TEI of Thessaloniki
Thessaloniki, Greece
v13@v13.gr

Nikolaos Samaras
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
samaras@uom.gr

Vasileios Vitsas
Department of Informatics
TEI of Thessaloniki
Thessaloniki, Greece
vitsas@it.teithe.gr

***Abstract*—This paper presents a realistic method for exploiting an already known insecurity of the Congestion Control behavior of the Transmission Control Protocol that was originally pointed out in 1999 [11] and that affects all known TCP implementations. This insecurity exploits the fundamental assumption of TCP that the communicating remote end is trustworthy and is behaving correctly. We developed a methodology and an algorithm which we used to attack a web server and deceive it in transmitting with a constant rate of 900 Mbits per second. During the attack the server was incapable of reacting to the network congestion it caused.**

***Keywords*-TCP; congestion control; security; opt-ack**

I. Introduction

The Transmission Control Protocol (TCP) [8] is one of the core Internet protocols. Almost all applications that require reliable data transfers take advantage of its capabilities. In October 1986 the Internet had the first of what became a series of "congestion collapses" [6] which lead Van Jacobson and Mike Karels in proposing a method for "Congestion Avoidance and Control" [6] which added the flow control ability to TCP. Since then, great efforts were made to improve the behavior of TCP and make it more robust by trying to improve congestion control logic [2], reduce packet loss [7], [9], improve performance [5], [1] and further set the basis for future development [2].

Because of its wide acceptance and deployment, all these years TCP has been the target of sophisticated security attacks from third parties leading to the research and development of appropriate defense strategies [3], [13]. In 1999 Savage et al. identified three attack methods [11] that could be performed by remote end-points instead of third parties and coined the term "misbehaving receivers". In 2005 Sherwood at al. focused on the Optimistic Acknowledgment (opt-ack) attack [12] by illustrating the possibilities of its exploitation.

The opt-ack attack is based on the TCP assumption that the remote end is trustworthy. Existing Congestion Control Algorithms increase the transmission rate of TCP whenever there is direct indication of no packet-loss. Since this indication is solely based on the receipt of positive ACKs, it is possible that a misbehaving receiver sends those ACKs even when the transmitted data are lost.

In this paper we present a method for easy exploitation of this vulnerability and the experimental results of its implementation. We also prove that the Hypertext Transfer Protocol [4] (HTTP) can be effectively combined with this attack method. Even though the possibility of the opt-ack attack has been presented in the literature [11], [12], this is the first implementation that has been successfully utilized to perform attacks. The proposed algorithm is adaptive and can be used to attack both: (a) high bandwidth servers and (b) extremely loaded, low bandwidth servers where desynchronization issues occur. The pointed implementation is based on a simple custom made user-space program written in Python language and takes advantage of a set of opensource libraries. All these make the proposed approach very flexible and quite portable.

The rest of the paper is organized as follows: Initially we describe the problem in section II and the tool we developed in section III. In section IV we introduce an efficient attack method against HTTP servers and we conclude by presenting the results of a performed attack against a web server under our administration in section V.

II. Problem Description

TCP has always been developed as a method for exchanging data between trusted hosts. Very few researchers have been looking at the possibility that one of the two TCP connection endpoints is misbehaving. More specifically, congestion control has always been an issue of detecting network congestion and acting against it. No Congestion Control Algorithm has ever noted the possibility of being abused by the other end. One can be sure about this by looking at the "Security Considerations" sections of TCP related Request For Comments documents (RFCs) which list all known security issues [10] at the time of their writings. Furthermore, no RFC has ever been published to address this issue.

TCP Congestion Control Algorithms rely on the received acknowledgments (ACKs) to identify network congestion

and reduce the transmission rate. Network congestion is either identified by detecting packet loss [6] or by looking for the Congestion Experienced (CE) codepoint when Explicit Congestion Notification (ECN) [9] is being used. Currently the underlying network may only drop or mark packets to indicate congestion.

Since Acknowledgments report which segments the other end has received, the sender uses them to find out whether packet loss was encountered and to (a) retransmit lost segments and (b) reduce its congestion window size (cwnd), which results in a lower transmission rate. The transmission rate (R) of a TCP Sender is a function of the congestion window size (cwnd) and the actual Round-Trip Time (RTT) and is given by:

$$R = \frac{cwnd}{RTT}$$

The value of the congestion window size depends on the number of successfully delivered segments which are inferred by the received ACKs. Round-Trip Time (RTT) is determined at the sender's side by measuring the time it takes for an ACK to be received after its corresponding segment was transmitted. We easily conclude that the transmission rate is (currently) based only on the behavior of the remote end which is responsible for transmitting the ACKs. In 1999 Savage et al. identified three attack methods [11] that could be performed by remote end-points instead of third parties and coined the term "misbehaving receivers". In 2005 Sherwood at al. focused on the Optimistic Acknowledgment (opt-ack) attack [12] by illustrating the possibilities of its exploitation.

The opt-ack attack is based on the TCP assumption that the remote end is trustworthy. Existing Congestion Control Algorithms increase the transmission rate of TCP whenever there is direct indication of no packet-loss. Since this indication is solely based on the receipt of positive ACKs, it is possible that a misbehaving receiver can send those ACKs even when the transmitted data are lost.

During the last years Internet connection speeds have greatly increased and Autonomous Systems with 1Gbps[1] or more of Internet connectivity are now a commonplace. Even though end-systems have increased their Internet connection speeds, backbone connections did not improve their speeds with the same rate. The current ratio of edge link speeds versus backbone link speeds and its trend indicate that during periods of time ISP backbones become the actual bottleneck. During the coming years, it may be more possible than ever, since the adoption of Congestion Control by TCP, for congestion issues to occur again.

We thus consider the possibility that Internet backbones may become badly congested by malicious end-nodes who flood at high speeds. This is caused by the fact that the correct Internet behavior is currently based on the assumption that end nodes react on congestion in a very drastic way which isn't always adequate.

Apart from Savage et al and Sherweed et al, there is no other concern or alert by vendors and security organizations regarding this issue. This statement is based on a conversation that took place at the IETF Maintenance and Minor Extensions (tcpm) mailing list, some older archives of that list and the lack of documents that cite [11] and actually address this issue.

Our research of this TCP flow lead to a method that is able to cause an unsuspected web server to perform flooding using its maximum transmission rate without having indication of network congestion. In contrast with the work that was presented in [12], we've implemented a simple flooding application working under the Linux Operating System, without modifying the underlying kernel. This application was repeatedly able to cause a web server (under our administration) to flood the Internet for 5 minutes (until it was interrupted) with 900Mbps of traffic using just a small portion of the actual uplink bandwidth of a DSL line operating at 8Mbps/1Mbps (downlink/uplink).

III. Tool Description

To be able to test the validity of the opt-ack attack on the current Internet using real-world victims we created a fully function testing attack suite. Using the Linux operating system and the Python language, we implemented a primitive TCP/IP stack. This stack is able to negotiate TCP Window Scaling [5] which is required for high bit rates. For convenience we also implemented the Timestamp Option to be able to perform Round-Trip-Time Measurement (RTTM) [5].

The python program uses the ImPacket library for constructing custom made IP and TCP PDUs, a raw socket for transmitting the custom made packets and the pcap library for detecting and effectively receiving data that were transmitted by the server.

Since the program runs on a network-enabled operating system, each incoming packet will be also received by the underlying kernel. As dictated by the TCP specification [8] the operating system's network stack will respond with RSTs to incoming TCP segments that are not related to a legitimate connection. For the program to be able to function properly we need to add a firewall rule that drops all incoming packets from the web server's IP address, or just the related ones. This prevents the local operating system from replying with RSTs, while letting the program capture incoming packets (packets are captured by the pcap library a lot earlier than the firewall checking and as soon as they arrive at the local station). Before performing the attack we also need to find a file to be requested from the server.

[1]We use the term "Bps" (with capital B) to indicate bytes per second and "bps" (with lowercase b) to indicate bits per second.

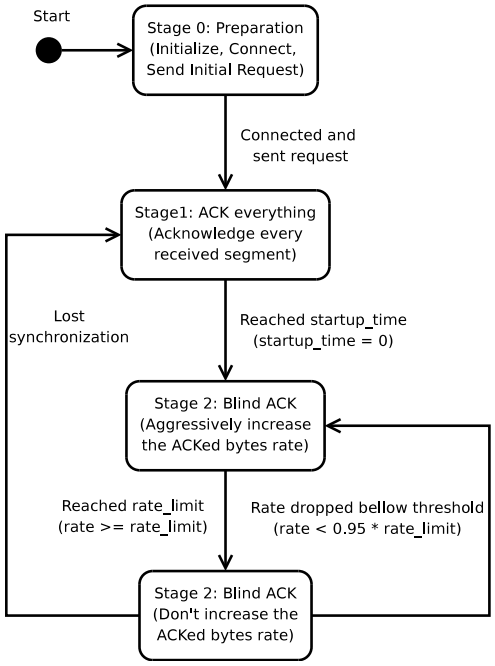

Figure 1. Algorithm State Transition

IV. Attack Description

To perform this kind of exploitation one needs a Linux client. He also needs to locate a web server with a fast Internet connection, capable server-side hardware and a large (10MB or more) file to serve, even though we are able to successfully perform this attack using 1MB files by taking advantage of HTTP/1.1.

A. Client Side

The actual TCP exploitation consists of an initial preparation step (stage 0) and a two-stage attack. During preparation the connection is established and an initial request is sent to a web server. While in stage 1 all incoming segments are immediately acknowledged, even though some data may be lost. In this stage the client synchronizes itself with the server. After a predefined period of time, long enough to achieve synchronization, stage 1 finishes and stage 2 begins. While in stage 2, ACKs are sent without waiting for data to be received.

Figure 1 illustrates the state transition of the actually implemented algorithm, which is a bit more complex than the one described here. The extra complexity goes beyond the scope of this paper and is there just to handle corner cases where the rate is reduced or synchronization is lost.

Stage 0: Preparation

During preparation the client performs the three-way handshake and connects to the web server. For our cause we prefer to use the very common HTTP protocol. HTTP servers reply to appropriate requests that are sent to them by clients who connect to server-side TCP port 80. Each HTTP session includes at least one client request and at least one server response. The client request is performed using one of the HTTP methods (GET, PUT, POST, HEAD, etc...). For our needs we're performing an HTTP GET request that asks the server side to transmit an existing file.

HTTP 1.1 [4] supports a keepalive mechanism which dictates that the server side may be willing to wait for more requests upon completion. This way a client is able to retrieve multiple objects from the server side without opening multiple connections. This method greatly speeds up the user experience of the world wide web and reduces the overall protocol overhead. As explained later the attack benefits a lot from this feature so we always use HTTP/1.1.

We may also want to use the range capability of HTTP as specified in section 14.35 of RFC2616 [4]. Using the Range header a legitimate client is able to request only part of a file, which is very useful for resuming downloads. For our purposes we take advantage of a "weakness" of Apache and IIS web servers. Both of them accept requests that specify the same byte range multiple times and are willing to serve it as many times as it was requested, even though this is not required for normal use. This solves the disk I/O problems that are mentioned in section IV-B.

A sample request looks like:

```
GET /~v13/debian-40r3-i386-CD-1.iso HTTP/1.1
Range: bytes=1-100000000,1-100000000,
  1-100000000,1-100000000
Host: host.example.org
```

Here we perform an HTTP 1.1 request and we ask from the server side to transmit four times the first 100MBytes of a file. From our tests we concluded that it is possible to request the same range for more than 500 times without encountering problems. This way we eliminate the need of finding a large file at the server side.

After performing the request the client goes to stage 1.

Stage 1: ACK everything

In this stage the client receives packets from the server side and unconditionally responds with ACKs. This achieves the required synchronization between the transmission and acknowledgment rate. Since ACKs are cumulative we are overlooking lost packets and indirectly send acknowledgments for them too. While in this stage we're able to increase the rate up to the point where at least one segment is received every no more than one window of data. During that time the rate slowly increases but the maximum achievable rate is relatively small and hard to reach.

While in this stage we measure differences in incoming sequence numbers and determine the transmission rate of the other end, regardless of the rate that we're able to receive data. We keep working like this for a predefined period of time (*startup_time*) and at the end we're left with a measured rate of $R0$.

We note here that the whole algorithm is clocked by received segments and that we choose not to send an ACK for each one of them. We transmit an ACK every *ack_interval* packets to reduce our CPU usage and upload bandwidth consumption. We also avoid to send duplicate or out-of-order ACKs to inhibit the sender from reducing its transmission rate. A suggested value for *ack_interval* is 3 but it can be further increased or decreased to meet one's needs.

After the startup period the client enters stage 2.

Stage 2: Blind ACKs

At this point we stop depending on the received data from the other end and we start sending ACKs assuming that it transmits with rate R_n where n is the iteration number and $R_0 = R0$. One can easily infer that at iteration n we only need to increase the ACK number by $dt * R_{n-1}$ where dt is the time since the last transmitted ACK. The time of the last transmitted ACK is the time of the last iteration (t_{n-1}):

$$dt = t_n - t_{n-1}$$

This is adequate to keep the rate at a constant level. The ACK number for iteration n is given by:

$$ACK_n = ACK_{n-1} + R_{n-1} * dt \quad (1)$$

where ACK_n is the Nth transmitted ACK, R_{n-1} is the last measured transmission rate and dt is the time since the last ACK was sent.

Since we want to further increase the transmission rate we introduce a boost value which is periodically added to the ACK number. We choose a predefined assumed RTT value and the number ($cwnd\_inc$) of segments of size MSS we want to increase the other end's Congestion Window (*cwnd*) every RTT. Boost is given by:

$$boost = cwnd\_inc * MSS * \frac{dt}{RTT}$$

where MSS is the Maximum Segment Size. For servers that use aggressive TCP implementations, $cwnd\_inc$ can have values greater than 15. For low-bandwidth servers, servers with high network load and servers with conservative TCP implementations, $cwnd\_inc$ can be set to 1. When considering boost, equation 1 becomes:

$$ACK_n = ACK_{n-1} + R_{n-1} * dt + boost$$

We further add an upper rate limit where we stop adding boost. This limit is chosen so that the server will not reach its maximum local link transmission rate, since it will encounter local congestion. This limit is determined by preliminary test runs where we observe the point where the algorithm stalls. Most of the time this means that the server stops increasing its transmission rate and the algorithm looses its synchronization and we use the rate of that moment as the maximum rate. Typical values are: 105000 KBps for servers with 1Gbps Ethernet connection, 30000 KBps for servers with 1Gbps Ethernet connection without very capable hardware, 10000 - 12000 KBps for servers with 100Mbps Ethernet connection and 5000 KBps for network-loaded servers.

B. Server Side Concerns

When requesting from a server to transmit at rates near 1Gbps, pushing it to its limits, we have to face some other real-world problems too. For a server to be able to serve a file at a constant rate of 1 Gbps or 125 Mbytes/second, it has to be able to perform disk reads at that rate. In most cases this is not possible and the server side will stalls waiting for I/O. We can circumvent this by selecting to transfer a file like a 650MB CD iso image which is large enough to be transmitted for more than 3-4 seconds using high bit rates, but small enough to be stored in memory cache or by using the range weakness described in section IV-A to perform a request of an equivalent total length.

Before we begin the attack we also need to download the file once. This will force the server to read it from the disk and have it immediately available for serving it from its memory cache. We then take advantage of the keepalive mechanism that was described earlier and keep performing the same request indefinitely. To achieve this we need to use a counter ($count$) at the client side to count the transmitted bytes. Whenever this counter reaches the approximate file size ($reinit$), a new GET request is sent to the server. This way the server is always busy retransmitting the same file again and again, forcing its operating system to keep it in memory cache (since it is being requested every few seconds) and serving it without ever closing the connection.

When performing transfers at speeds near 1 Gbps or 125 MBps, a large amount of data is always on-the-fly. For $RTT = 200ms$ and $R = 100000KBps$ there are always 20MBytes of data on the intermediate network. This means that we have to be proactive and initiate a new transfer a little bit earlier. Assuming that a request needs $RTT/2$ time to reach the other end, we transmit a new request whenever $count$ reaches $reinit - (R * RTT)$. This way we are ahead of our time by at least $RTT/2$ and we do not allow the sender side to stall waiting for our next request since this could reduce its transmission rate.

V. Results

Using this technique one can achieve speeds of up to more than 900Mbps after a very short period of time (20-60 seconds) and force the server side in constantly transmitting almost at its maximum rate. The actual time it takes to reach the maximum speed depends on the parameters that were used. Since this approach can be adjusted by parameters to become very modest in the way it increases the transmission rate, it is not error prone and has proved to be very efficient.

```
Rate: ACK: 107508.585 KBps, 256.93 ACKs/sec,
    420.86 MB Transferred, RTTM: 0.340088
Rate: ACK: 107755.252 KBps, 257.65 ACKs/sec,
    526.24 MB Transferred, RTTM: 0.343079
Restart
Rate: ACK: 107595.764 KBps, 256.92 ACKs/sec,
    0.00 MB Transferred, RTTM: 0.344116
```

Figure 2. Sample Output

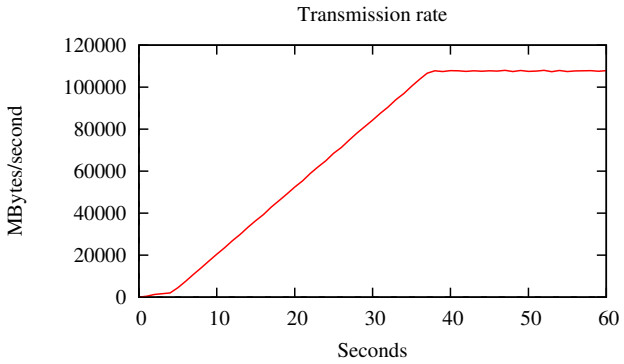

Figure 3. Transmission rates

A sample output is shown in Figure 2. In each "line" we see the transmission rate in KBytes/second based on the difference of transmitted ACK numbers, the transmitted ACK rate, the MBytes of the file that were ACKed (it resets every 620 MBytes) and the RTT as it was estimated by the Timestamps option (RTTM). Lines are printed every 1 second and the "Restart" line indicates that a new GET request was sent to the server to initiate another file transfer.

Figure 3 shows how the transmission rate increases in time and stabilizes when it reaches the pre-set upper limit. The graph was produced using modest parameter values and illustrates the two stage period and the achieved constant rate. Stage 1 lasts 5 seconds (0-4) and manages to synchronize the algorithm. Stage 2 lasts 33 seconds (5-37) and deceives the server side in increasing its transmission rate up to a little more than 107 MBytes/second (more than 897 Mbps). After that the algorithm stabilizes to the achieved rate until interrupted.

The indicated rate is determined from the difference of the ACKed bytes and thus it is the actual throughput of the transmitted TCP data stream. Assuming that the Path MTU (PMTU) is 1500 bytes (worst case scenario) and that IP and TCP headers consume no more than 40 bytes (worst case scenario too), the actual rate becomes $897 * 1500/1460 = 921$Mbps. This is the rate of the bytes that the server side floods the Internet with. The actual transmission rate at the local link of the server is determined by considering the overhead of the Ethernet protocol too and it is $921 * 1518/1500 = 932$Mbps.

A. Network congestion and observations

Because of the way the flooding is performed, network congestion will occur at the nearest to the server side congested link. After a short period of time all TCP connections that use this link will reduce their rate and they will almost pause. The attacked server will keep transmitting at its maximum rate occupying a larger portion of the congested link's bandwidth. Since this link will be almost exclusively used by the attacked server, the largest possible portion of the generated traffic will pass to the next congested link, propagating the problem to the nearest physical bottleneck.

High transfer rates can also cause high loads to firewalls and some low-end intermediate routers. Poor cabling and low-end hardware is also stressed. On one tested case networking problems occurred because of physical errors on an intermediate 1Gbps copper-based Ethernet connection. Those problems effectively made the attacked web server partially unavailable by disturbing the routing procedure of intermediate routers and causing frequent reroutes. On another case an intermediate low-end Linux-based firewall suffered kernel panics for unknown reasons.

During our research we have done a series of observations:

1) Many deployed web servers have the range weakness.
2) About half of the examined sites support the keepalive mechanism.
3) Akamai servers do not support the HTTP keepalive mechanism and thus they reduce the potential of exploiting them. Also, they aren't capable of transmitting at very high rates. Major companies like Microsoft and Sun use Akamai technologies to provide content and thus they aren't immediately vulnerable.
4) By using $cwnd\_inc = 1$ and high RTT values it is possible to exploit even long-distant, low-speed, congested servers.
5) Using this method one can determine the server side link speed.
6) Google is vulnerable.

Also, as expected, we found that proxy servers are vulnerable to this attack too. For example, the Squid cache (the most common and widely used HTTP caching proxy server) fully supports persistent connections (as it calls them) and defaults to "on" with a request timeout of 30 seconds and a maximum connection lifetime of 1 day. The availability of open-proxies (i.e. proxies that can be used by anyone, even outside of the network provider) only worsens the situation.

VI. Conclusions

As mentioned in [12] the threat of a distributed denial-of-service (DDoS) attack using this technique actually exists and should not be taken lightly. We have implemented a malicious client using a popular language like Python and proved that this attack can be effectively performed by using

only user-space tools (i.e. there is no need to modify the underlying Operating System). Also, since the number of available high-speed end-points is quite large, one needs to find 10-100 such servers to perform a Distributed DoS (DDoS) attack to an ISP just by having control of a couple of DSL lines.

A large portion of the world-wide deployed web servers can be abused and tricked into flooding the Internet using their maximum transmission rate. Some major sites like Google's seem to be vulnerable while others like Microsoft's that use Akamai technologies are not. This attack may also be used to perform Denial of Service (DoS) attacks to web-servers and Autonomous Systems (ASs) with lesser hardware/infrastructure.

REFERENCES

[1] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 2414 (Experimental), September 1998. Obsoleted by RFC 3390.

[2] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[3] S. Bellovin. Defending Against Sequence Number Attacks. RFC 1948 (Informational), May 1996.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[5] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

[6] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[7] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[8] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[9] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001.

[10] E. Rescorla and B. Korver. Guidelines for Writing RFC Text on Security Considerations. RFC 3552 (Best Current Practice), July 2003.

[11] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. Tcp congestion control with a misbehaving receiver. *SIGCOMM Comput. Commun. Rev.*, 29(5):71–78, 1999.

[12] Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud. Misbehaving tcp receivers can cause internet-wide congestion collapse. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 383–392, New York, NY, USA, 2005. ACM.

[13] J. Touch. Defending TCP Against Spoofing Attacks. RFC 4953 (Informational), July 2007.