Analyzing of RESPIRE, a novel approach to automatically blocking SYN flooding attacks

ANDRÁS KORN, JUDIT GYIMESI, DR. GÁBOR FEHÉR

Budapest University of Technology and Economics, Department of Telecommunication and Mediainformatics, HSNLab

korn@chardonnay.math.bme.hu, gj309@hszk.bme.hu, gume@eik.bme.hu

Reviewed

Key words: attack, counting, flooding, syncookies

A few years ago, numerous major web sites were successfully brought down using an attack called SYN flooding. A number of methods for combating SYN floods have been proposed, many of which are widely deployed. In this paper, we describe a possible enhancement to some of these techniques; a way to automatically detect, isolate and filter SYN floods while conserving resources on the victim.

1. Introduction

The TCP SYN attack is made possible because establishing a TCP connection involves a so-called three-way handshake. The client starts the connection by sending a packet with the SYN flag set; it also specifies an Initial Sequence Number (ISN). The server replies to this with a packet that has both the SYN and the ACK flags set; it contains an acknowledgement for the ISN of the client and the ISN of the server. The connection is finalized when the client replies to this message with an ACK packet that acknowledges the ISN of the server.

In order for the server to be able to verify that the final ACK packet is indeed a reply to the SYN ACK, it has to compare the acknowledged sequence number with the ISN it gave the client; thus, it is necessary to establish a state when the SYN ACK packet is sent and to maintain it for some time: either until the final ACK arrives, or until it times out.

The attacker thus merely needs to send copious amounts of SYN packets (perhaps using spoofed source addresses). He or she ignores the SYNACK packets of the victim and never finalizes the connection. After a while, the finite connection backlog of the victim will be full and no further TCP connections to the attacked port will be possible.

2. Existing solutions

Some vendors (e.g. Cisco) offer routers that claim to offer protection against SYN floods. Some of these proxy the TCP handshake: they only send SYN packets to the protected server if they already received the final ACK. For the connection to work, the sequence numbers must be mangled on each subsequent packet of the session (because the router had to choose an initial sequence number for the connection, and the ISN of the server is bound to be different). These routers typically also have shorter timeouts on half-open connections and thus are indeed less vulnerable to SYN floods. It is important to note however that these approaches don't solve the actual problem, they merely increase the cost of a successful attack. It is still necessary to allocate finite resources (memory) for each connection. The only difference is that the attacker has to deplete the memory of the router, not the server.

Another suggested solution was to randomly drop SYN packets using a RED scheme [6]. Like shorter timeouts, RED also only makes the attack more expensive, but not impossible.

There are very general and thus somewhat heavyweight ways of dealing with flooding and congestion in general; one of these is described in [1].

An ingenious and widely deployed defense against SYN floods are TCP syncookies [3]. Syncookies work by sending a carefully crafted, cryptographically strong ISN back to the client in the SYN ACK packet, so that the ACKed sequence number in the final ACK packet is enough to validate the connection. No state is established and no memory used until the final ACK arrives. It is unfeasible for the client to guess a valid ACK sequence number and thereby spoof a connection without receiving a SYN ACK packet from the server first.

Detecting SYN floods is a different problem. One of the proposed solutions is described in [4] – while a stateless, "dumb" device does have its merits, the problem with this particular approach is that it can only help filter the flood if the device is located near the attacker. This means it would have to be deployed at every ISP worldwide in order to be useful. Failing that, the device can only detect that a flood is in progress but can't tell us who the perpetrator is.

3. Problems with syncookies

However, using syncookies has drawbacks. First of all, a connection established using syncookies cannot use large windows and can only use a fixed set of Maximum Segment Size (MSS) values. Second, syncookies take

time to compute: Bernstein suggests using the Rijndael algorithm to generate the ISN. Third, they magnify the effect of the SYN flood by responding with a flood of SYN ACK packets – possibly to unwitting third parties, if the flood uses forged source addresses. Thus, syncookies can actually make the situation worse by allowing “bounce flooding”.

Therefore, even though syncookies ensure continued operation of a service even when under attack, it still makes sense to use a packet filter to prevent the offending SYN packets from reaching the server at all.

The approach presented in this paper is complementary to syncookies. The cookies can ensure that the service remains available while the *RESPIRE (Resource Efficient Synflood Protection for Internet Routers and End-systems)* mechanism reacts to the flood and blocks it; however, as shown below, reaction times are so short that syncookies are not strictly required.

4. How RESPIRE works

In contrast, our approach requires no additional data-gathering equipment to be deployed. Rather, it makes use of the data the victim itself must collect anyway in order to be able to provide TCP service.

The victim has a plethora of useful information we can use to determine whether we are under a SYN flooding attack; for example, we probably are if any of the following conditions are met:

- the number of incoming SYNs per second exceeds a threshold;
- a TCP backlog queue gets filled, so we have to start sending syncookies;
- the number of half-open connections exceeds a threshold;
- there is a disproportional difference between the number of SYN ACK packets sent out and ACK packets received.

RESPIRE as described here relies primarily on the last heuristic, but using a combination of all of the above is possible with minimal modifications.

Note that it would be possible to compare the number of arriving SYN packets to the number of inbound connection-finalizing ACK packets. However, in order to identify ACK packets that are indeed the last packet of a handshake, we need to track all TCP connections anyway, which involves analyzing the SYNACK packet and recording its ISN. Based on this information, we could reconstruct the SYN anyway, so processing the SYN packets separately seems redundant. However, in order for us to be able to rely on counting outbound SYNACK packets, the victim needs to be able to respond to a sufficient number of SYN flood packets with SYNACKs. Syncookies guarantee this ability, but if they can’t be used for some reason, we must choose a backlog size that allows enough SYNACK packets to be sent for RESPIRE to identify the attackers before the backlog fills up. If this cannot be done, we can still count inbound SYN packets instead of outbound SYNACKs, but still need to process outbound SYNACK packets as well because we need their ISN.

So, to sum it up: it makes sense to count inbound SYNs instead of outbound SYNACKs if the protected server can use neither syncookies nor a sufficiently large backlog queue.

When we are under a SYN attack, the best we can do is to ignore the SYN packets of the attacker. The simplest way to accomplish this is to set up firewall rules that block SYN sent by the attacker; this means that our most important objective is identifying the address(es) the attacker uses. We could only do better than this by “pushing” the filtering towards the attacker along the network route his packets traverse towards us using *pushback* [5] or a similar mechanism.

Glossary

A.B.C.D/E
This is a shorthand notation for an IP subnet where the first E of 32 address bits identify the network, with the remaining bits identifying a node within that network. For example, the Budapest University of Technology and Economics uses the 152.66.0.0/16 network. “E” is commonly referred to as the “size” of the network. The smaller E is, the more nodes the network contains.

ACK
One of the flags used in TCP. Indicates that the packet contains an acknowledgement of previously received data.

cookie
Used to denote cryptographically generated data that is used in authentication.

DoS
Abbreviation of “Denial of Service”. DoS attacks try to disable or sabotage a service.

SYN
One of the flags used in TCP. If set, the packet is referred to as a “SYN packet”. The TCP handshake starts with the client sending a SYN packet.

SYNACK
The second packet of the three-way TCP handshake is commonly called a “SYNACK packet”. It has both the SYN and the ACK flags set.

port
A two-byte endpoint identifier that is used by TCP and UDP to distinguish between network flows related to a single IP address.

RED
Random Early Drop. A congestion control mechanism that avoids congestion by dropping some packets before the network becomes congested.

spoofing
Forgery (of the source address of a packet).

sequence no.
Every data unit sent using TCP has a sequence number: basically the number of bytes transmitted so far plus a random offset determined at connection setup. The random offset makes connection forgery more difficult.

Historically, it used to be possible to forge just about any source address on a packet, so isolating the sources would not have been possible. By now, however, most networks have reverse path filters or are using other mechanisms to filter packets that are obvious fakes; therefore, an attacker can typically only forge addresses within one (or a handful of) class C network(s). We note that RESPIRE fails if the attacker can spoof any source address; in fact, it can exacerbate the situation by blocking legitimate clients in an attempt to block the attacker. Combining RESPIRE with spoof detectors like hop-count filtering [2] can significantly reduce this risk.

5. Anatomy of a SYN Flood

The typical attack scenario today is that the attacker has access to a number of computers compromised previously and now under his control – commonly referred to as "drones" – in several subnets around the world, and instructs all of them to launch an attack in concert, effectively mounting a distributed denial of service (DDoS) attack. To make filtering the packets more difficult, the drones use spoofed addresses, but every address is within the same netblock as the real address of the drone; otherwise, the edge router of the netblock would discard the packets.

Note that if the magnitude of the SYN flood is sufficient to flood the entire downlink of the victim, the attack is no longer a SYN attack but a generic bandwidth depletion attack that happens to use SYN packets; it is not our goal to deal with this scenario here.

Identifying the attacker

As mentioned earlier, we assume that during a SYN flood, the ratio of the number of outgoing SYN ACK packets to the number of incoming handshake-finishing ACK packets is going to be much larger than one. Note that most SYN ACK packets that go unacknowledged are sent to the attacker; thus, we can identify the attacker by finding the subnet with the most outgoing SYN ACKs per incoming ACKs.

A naive way of doing this would be to count the SYN ACK and ACK packets going to and coming from each class C subnet in a large table with 224 (16.7 million) entries. It is easy to see that this would be grossly inefficient; most of the counters would be zero, and most of those that are positive would only indicate benign behaviour. Finding the attacker would require looking at every entry in the table.

6. RESPIRE in detail

MULTOPS [7], the algorithm RESPIRE is loosely based on, addresses this problem by storing the counters in an efficient, dynamically expandable hierarchical data structure that exploits the hierarchical nature of IP space: a 256-ary tree is constructed to hold the counters.

The root of the tree contains two counters, initialized to zero, and 256 pointers, initialized to NULL. One of the counters, *Synack_Out*, counts the SYN ACK packets leaving the system. The other counter, *Ack_In*, counts the valid ACK packets (ones that finish TCP handshakes) entering the system.

After at least *Synack_Min* SYN ACK packets have been sent, the counters are consulted after each further *Synack_Num* SYN ACK packets are sent out. The tree structure makes this a relatively cheap operation to carry out. Because the number of tree levels is at most four, four divisions and comparisons and eight increments must be carried out per SYN ACK packet. For this reason, we recommend setting *Synack_Num* to one.

Sites with very large amounts of traffic can reduce the overhead by increasing *Synack_Num* at a small cost in flood detection speed and accuracy. Instead of deterministic sampling, stochastic methods can be used, or *Synack_Num* can be adjusted dynamically based on the amount of traffic received; however, these variations have no impact on the fundamental operation of the algorithm.

If the ratio of *Synack_Out to Ack_In* exceeds the value of the parameter R_{max} (a value of 1.5 or more is recommended), then in the last sampling period, the number of outgoing SYN ACK packets outnumbered the number of incoming ACK packets at least 1.5 to 1. This should not happen under normal circumstances, so we assume that we are under attack.

If we are under attack, we begin expanding the tree. For each *Synack_Num* subsequent outgoing SYN ACK or incoming ACK packet, we note the remote IP address A.B.C.D. If the root node pointer A is NULL, we allocate a new node with the same structure as the root node and link it to root→A. All SYN/ACK traffic associated with A.0.0.0/8 is from now on counted in both the root node and in the counters of root→A.

If root→A already exists, we check if
A→*Synack_Out* ≥ *Synack_Min*[L1] and if

$$\frac{\text{root}\to A\to\texttt{Synack_Out}}{\text{root}\to A\to\texttt{Ack_In}} > R_{\max}$$

If so, A.0.0.0/8 is probably one of the sources of the attack. We "zoom in" further by creating A→B if it doesn't already exist and so on until A→B→C exists.

The *Synack_Min* parameter can be different for each tree level. Decreasing the limit on the lower levels makes attack isolation faster but slightly less accurate. To compensate this, it would be possible to increase R_{max}. We plan to investigate such fine-tuning possibilities in a future paper.

If A→B→C exists, has at least *Synack_Min*[L3] SYN ACK packets associated with it and the ratio of its counters exceeds R_{max}, A.B.C is assumed to be an attacking subnet and is blocked, i.e. no further incoming SYN packets are accepted from A.B.C.0/24.

How this blocking is done is beyond the scope of this paper. The possibilities include, but are not limited to:

– Adding the filter to TCP stack of the OS kernel.
– Using the built-in packet filtering mechanisms of the underlying operating system, if any.
– Using a mechanism like *pushback* [5] to request filtering from an upstream router.

Naturally, these blocks should be temporary. After *Block_Timeout* minutes (15 recommended), they can be removed. This value should be chosen so that it is slightly longer than the typical flood is expected to be; it is unwise to set it too high, because having too many packet filtering rules puts a strain on the device that does the filtering. Also, continuing to block a subnet after the flood is over could result in blocking legitimate clients.

Once we find and block an attacking subnet, we remove its node from the tree; since we blocked it, we won't be receiving any further packets from it anyway, and we subtract its package counters from the counters in its parent nodes. This is done so that in the parent nodes the packet ratios more closely reflect the expected new distribution, where packets from the newly blocked subnet will no longer enter the system. This allows us to more accurately decide whether there still are other attacking subnets.

Once every *Prune_Interval* (2 seconds in our simulation), we check if the tree contains suspicious nodes (with counter ratios in excess of R_{max}). If so, we zero their counters. We remove all other nodes from the tree, except, obviously, the root node.

Only zeroing suspicious nodes instead of removing them allows us to more quickly identify them as attackers during the next *Prune_Interval*, because we don't have to wait for *Synack_Min* packets to accumulate in their parent nodes as the lower level node already exists.

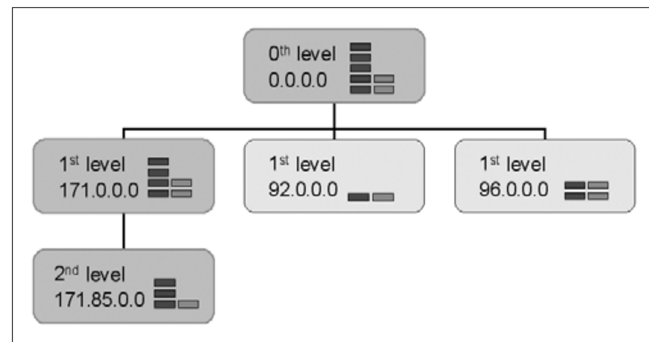
We zero the counters because we are only interested in ongoing flooding activity; we do not want past suspicious behaviour of a subnet to bias our future decisions. Unfortunately, this means that an attacker with a high number of distinct class C networks under her control can insinuate a "slow SYN flood" into the protected system; i.e. she can send less than *Synack_Min[L3]/ Prune_Interval* packets per second from each subnet, so that none of them are identified as individual flood sources and blocked, but their cumulative effect on the service is detrimental nevertheless.

In this case, we can classify the nodes as "above suspicion", "slightly suspicious" and "definitely suspicious". Nodes are above suspicion if their counter ratio is smaller than an R_{legit} value (1.1 or even 1.05). These nodes we can remove from the tree at *Prune_Interval* boundaries. Slightly suspicious nodes have counter ratios between R_{legit} and R_{max}. We zero the counters of these nodes but don't remove them yet. A node is definitely suspicious if its counter ratio exceeds R_{max} but it didn't yet accumulate *Synack_Min* packets.

We don't even erase the counters of such nodes. It is reasonable to expect that after a while the nodes associated with the attackers will satisfy the criteria of filtering.

Figure 1. below shows an example RESPIRE tree. The two columns inside the nodes represent the relative amounts of SYN ACK and ACK packets respectively (but not an exact count). The nodes 92.0.0.0/8 and 96.0.0.0/8 sent approximately as many ACK packets as we sent them SYN ACKs, so they are probably benign. The darker nodes on the left, 171.0.0.0/8 and 171.85.0.0/16, are the suspicious ones. Note that the root node is also "suspicious"; this is what tells us that we are under attack.

Figure 1. Example of a RESPIRE tree



RESPIRE Memory Usage

In order to avoid memory exhaustion attacks against RESPIRE, the total number of tree nodes must be limited. One node occupies 2x64 + 256x64 bits: the two counters and the 256 pointers, assuming a 64-bit architecture. This adds up to 2064 bytes, or half that, about one kilobyte on more common 32-bit computers. The maximum number of nodes that could be created if no limit were enforced is 16777216 + 65536 + 256; thus, the total amount of memory used by the tree structure could increase to up to about 32.5 gigabytes (half this on 32-bit architectures), which is impractical to store and manage.

Our simulation showed that more than 150 nodes are seldom required even when the attackers command rather large address spaces. If we assume an unrealistic case where 200 different class C networks are used to flood the victim, and these all reside in different A blocks, only 600 nodes would have to be allocated, adding up to slightly more than one megabyte in size. Thus, limiting the amount of nodes to about 500 seems safe.

What to do when the limit is reached? If a new node is to be created beyond the 500th, find the least suspicious node under the root node and remove it and its children. If the root node only has one branch, continue the search on level A; obviously the single A node must have more than one branch, or we could not have 500 nodes in total. (More sophisticated methods could be used to find nodes to delete, but they would be more expensive.)

7. RESPIRE reaction time

In this section we will try to estimate the reaction time of the algorithm using mathematical methods. Let us first assume that we are dealing with a single attacking class C subnet. The calculations can be generalized to apply to more complicated cases, except “slow floods”, which were discussed earlier.

In cases where multiple class C subnets are attacking, we can aggregate the times needed for each attacking subnet to be banned. This will be the worst case, because usually we should be able to ban several subnets in one go.

We assume that the attacking SYN packets arrive with an intensity of Ψ packets/sec. The legitimate traffic can be described as a λ parameter Poisson process, since legitimate users are independent. If we assume that burstiness is minimal, outbound SYNACK and inbound ACK packets likewise resemble Poisson processes, because the time needed for the computer to compose a SYNACK packet from an incoming SYN requests is approximately constant. This way, the Round Trip Time (RTT) does not cause a significant error in this approximation.

Even though the ACK packets arriving in a time interval are not necessarily the replies to the outgoing SYNACK packets of the same interval, the expected value of their number should be almost equal; with Poisson processes, we can expect a similar number of events in intervals of the same length, regardless of when the intervals start.

Let Δt be the time the attack begins after a *Prune_Interval* boundary. Two conditions are tested by the algorithm:

$$\frac{\lambda_l \cdot \Delta t + \psi_t \cdot \Delta t}{\lambda_l \cdot \Delta t} \geq R_{\max}$$

and

$$\lambda \cdot \Delta t + \psi_t \cdot \Delta t \geq synack_min_{root}$$

The first inequality is independent of Δt; thus, if the above assumptions hold, we recognize an attack as soon as *Synack_Min* is exceeded. The condition of detection thus is:

$$\Psi_t \geq (R_{\max} - 1) \cdot \lambda_l$$

Thus the time needed for each tree level is:

$$\Delta t'_{level} = \frac{synack_min_{level}}{\lambda_{level} + \psi}$$

Naturally, detection takes longer if, while counting, a *Prune_Interval* ends, because all counters are zeroed. Fortunately, counting can be resumed at the same tree depth we left off, because parent nodes are not erased. Using the indicator function *I{A}* which returns 1 if condition A is met and 0 otherwise, both cases (i.e. crossing an interval boundary or not) can be written in one equation. The condition tests whether the *Prune_Interval* the counting started in is different from the one it is supposed to end in. More than one boundary cannot be crossed; if the criteria of detection are met, we detect the flood in either one or two intervals.

$$\Delta t_{root} = I\left\{\left\lceil \frac{\Delta t_t}{\Delta t_p} \right\rceil < \left\lceil \frac{\Delta t_t + \Delta t'_{root}}{\Delta t_p} \right\rceil\right\} \cdot \left(\left\lceil \frac{\Delta t_t}{\Delta t_p} \right\rceil \cdot \Delta t_p - \Delta t_t\right) + \Delta t_{root}'$$

Simplifications are possible by recognizing that if detection cannot be finished in the interval it started in, the time remaining until the next boundary is less than $\Delta t'_{level}$. In the next interval, we start counting again, and will take approximately $\Delta t'_{level}$ seconds to identify the attacker. The time spent at a tree level can thus be estimated by:

$$\Delta t_{level} \leq 2 \cdot \Delta t'_{level}$$

The probability of crossing an interval boundary is smaller if we set the interval length larger. A compromise must be found: the desire to only detect on-going attacks requires the interval to be small, whereas reaction times are better if intervals are longer.

When the root node indicates attack, we start building the tree. This procedure, along with the examination of the nodes, their manipulation, counting etc. do not cause a relevant time overhead, because packet transmission times are typically several orders of magnitude larger.

In order to better estimate the time spent at each tree level, we need to introduce a parameter “a” that indicates what fraction of the legitimate traffic originates from a given subnet as we move down the tree. Assuming that every subnet is responsible for an equal portion of the whole traffic would mean that packets from a specific class A subnet make up only about 1/256th of all the incoming SYN requests. For a class B subnet, the amount would be $1/256^2$, for a class C subnet, $1/256^3$. Naturally, this will not be true in practice. One reason is the distribution of IP addresses. At the A level, a significant portion of the address space is reserved for special purposes. The B and C address spaces are also unequally used, but the situation is not as bad as at level A. Thus an approximation where we use the parameter only at the level A, and the lower levels are taken to be homogeneous, appears to be acceptable. The value of “a” will vary, but probably be somewhere between 16 (local server used mostly within a relatively small country) and 128 (busy global server).

Total reaction time is the sum of the time spent at each of the four levels. These, if we needn’t cross any *Prune_Interval* boundaries, and are using different *Synack_Min* values for each level, can be written as follows:

$$\Delta t'_{root} \geq \frac{synack_min_{root}}{\lambda_l + \psi_t}$$

$$\Delta t'_A \geq \frac{synack_min_A}{\frac{1}{a} \cdot \lambda_l + \psi_t}$$

$$\Delta t'_B \geq \frac{synack_min_B}{\frac{1}{a \cdot 256} \cdot \lambda_l + \psi_t}$$

$$\Delta t'_C \geq \frac{synack_min_C}{\frac{1}{a \cdot 256^2} \cdot \lambda_l + \psi_t}$$

If we also wish to account for the possibility of crossing interval boundaries, the equations become more complex:

$$\Delta t_{root} = I\left\{\left[\frac{\Delta t_t}{\Delta t_p}\right] < \left[\frac{\Delta t_t + \Delta t'_{root}}{\Delta t_p}\right]\right\}\cdot\left(\left[\frac{\Delta t_t}{\Delta t_p}\right]\cdot\Delta t_p - \Delta t_t\right) + \Delta t'_{root}$$

$$\Delta t_A = I\left\{\left[\frac{\Delta t_t + \Delta t_{root}}{\Delta t_p}\right] < \left[\frac{\Delta t_t + \Delta t_{root} + \Delta t'_A}{\Delta t_p}\right]\right\}\cdot\left(\left[\frac{\Delta t_t + \Delta t_{root}}{\Delta t_p}\right]\cdot\Delta t_p - \Delta t_t - \Delta t_{root}\right) + \Delta t'_A$$

$$\Delta t_B = I\left\{\left[\frac{\Delta t_t + \Delta t_{root} + \Delta t_A}{\Delta t_p}\right] < \left[\frac{\Delta t_t + \Delta t_{root} + \Delta t_A + \Delta t'_B}{\Delta t_p}\right]\right\}\cdot\left(\left[\frac{\Delta t_t + \Delta t_{root} + \Delta t_A}{\Delta t_p}\right]\cdot\Delta t_p - \Delta t_t - \Delta t_{root} - \Delta t_A\right) + \Delta t'_B$$

$$\Delta t_C = I\left\{\left[\frac{\Delta t_t + \Delta t_{root} + \Delta t_A + \Delta t_B}{\Delta t_p}\right] < \left[\frac{\Delta t_t + \Delta t_{root} + \Delta t_A + \Delta t'_C}{\Delta t_p}\right]\right\}\cdot\left(\left[\frac{\Delta t_t + \Delta t_{root} + \Delta t_A + \Delta t_B}{\Delta t_p}\right]\cdot\Delta t_p - \Delta t_t - \Delta t_{root} - \Delta t_A - \Delta t_B\right) + \Delta t'_C$$

The overall time needed obviously equals:

$$\Delta T = \Delta t_{root} + \Delta t_A + \Delta t_B + \Delta t_C$$

Let us consider the case where all legitimate SYN packets come from different class A subnets than the attacking ones. This gives us the worst case, since detection time depends only on the time needed to collect at least *Synack_Min* SYN packets – if the attack can be recognized at all. Under these circumstances we need not assume anything about the distribution of legitimate traffic among the subnets. We obtain:

$$\Delta t'_{root} = \frac{synack_min_{root}}{\lambda_l + \psi_t}$$

$$\Delta t'_A = \frac{synack_min_A}{\psi_t}$$

$$\Delta t'_B = \frac{synack_min_B}{\psi_t}$$

$$\Delta t'_C = \frac{synack_min_C}{\psi_t}$$

Reaction time when the algorithm crosses an interval boundary at each tree level can be written as:

$$\Delta T = 4 \cdot \Delta t_p - \Delta t_t + \frac{synack_min_C}{\psi_t}$$

Please note that although reaction time seems to increase as *Prune_Interval* increases, its expected value actually decreases, as the earlier equations indicate. The reason is that when the interval is longer, the probability of reaching the interval boundary decreases.

Let us now give a rougher, but more compact approximation. According to our earlier observations, the time needed at each level is less, than double the time that would be required if no interval crossing took place. Total reaction time is thus smaller than the time we get by assuming that every level needs as much time as the level that needed the most time.

$$\Delta T \leq 8 \cdot \max_{level}\left\{\frac{synack_min_{level}}{\lambda_{level} + \psi}\right\}$$

Note that reaction time decreases as attack intensity increases. Reaction is practically immediate in the case of extreme floods, which cause the most damage.

This corroborates the results of the simulations; see below.

8. Simulation results

In order to analyze the performance of RESPIRE, we also ran a simulation [8]. For the sake of completeness, we sum up the results in this paper as well.

We simulated a busy SMTP server that has 62.8 active connections and 12.6 new connections per second on average. 300 simulated terminals were used to represent legitimate clients that randomized their IP before each connection. We also planted 8 attackers into the system who flooded the server with SYN packets. Using these attackers we modelled a distributed SYN attack. The state machine implemented in the attackers was different from the ones in the normal clients. The attackers use spoofed source addresses that are uniformly distributed across an entire subnet, the base address of which is a random value. The subnet mask is chosen randomly between 16 and 24; attacks that use entire /16 subnets should be very uncommon in practice, but we wanted to be generous with the attackers in order to put RESPIRE to a harder test. Each attacker performs one SYN attack of random length and intensity.

Table 1. (on the next page) summarizes some of the numerical results of the simulation.

Note that attacks #5 and #6 appear twice. This happened because these attacks lasted longer than the timeout for the firewall rules (15 minutes), so after the rules expired, these attacks had to be blocked again. "Unfiltered packets" shows the number of flood packets that passed RESPIRE by before the filtering rules took effect. The other columns should be pretty self-explanatory.

Let us now compare simulation results with our mathematical predictions.

Attacker #3 has a subnet of 4096 addresses (16 adjacent class C networks). This means we will see 16 suspicious class C nodes in the tree, all descendants of the same class B node. If the attacker chooses the source addresses of his packets uniformly, each of these nodes will account for 1/16th of the total flood intensity, that is, 3660.19 pps. Down to level B we can act as if we only had a single attacking class C:

Attack No.	First packet (s)	Subnet size	Packet rate (pps)	Unfiltered packets	Reaction time (s)
1	80,780	32768	41274	22191	0,629
2	239,046	512	91938	505	0,011
3	764,754	4096	58563	1920	0,045
4	890,803	512	82013	505	0,011
5	1229,573	16384	39586	6766	0,244
6	2039,08	8192	40932	3535	0,101
5*	2129,699	16384	39586	6767	0,165
6*	2939,157	8192	40932	3535	0,113
7	4060,253	32768	88895	13231	0,193
8	4729,277	512	31267	505	0,021

Table 1. Attacker activity

$$\Delta T_{root} + \Delta T_A + \Delta T_B \le 6 \cdot \max_{szint=root,A,B} \left\{ \frac{synack_min_{szint}}{\psi_t} \right\} = 0,01$$

At level C, we obtain:

$$\Delta T_C \le 2 \cdot \frac{synack_min_C}{\psi_C} = 0,055$$

The total predicted time is thus 0.065 s which is a good estimate of the measured 0.045 s. The two methods produce comparable results.

Using the same methods, we can compute the predicted reaction times for all attacks: 0.635 s, 0.011 s, 0.65 s, 0.012 s, 0.338 s, 0.17 s, 0.032 s.

We needn't modify our upper estimate if several nonadjacent class C subnets start attacking at almost the same time. While this decreases the time needed for the root node to become suspicious, it doesn't influence the lower levels because the attackers reside in different class A nets. In our estimate, we used the time spent at the level where we spent longest, which certainly isn't the root node. Thus, the fact that the root node needs less time doesn't impact the rest of the calculations.

Naturally it can happen that the distribution of spoofed source addresses is non-uniform, which causes us to detect one attacking class C subnet before another. If some class C subnets use a substantially smaller attack intensity, total time taken can increase. Note however that in this case, the more damaging part of the flood has already been filtered, so it's acceptable to spend slightly more time blocking the rest.

9. Conclusion

In this paper, we introduced RESPIRE, one of several ways to combat SYN-floods. It appears to be a very lightweight solution that nevertheless filters SYN floods quickly and reliably. Additionally, it also reduces the amount of collateral damage a SYN flood can cause. Its memory requirements are very modest, rising above a few kilobytes only when under attack.

We suggest that RESPIRE be deployed alongside syncookies. A reference implementation for Linux is currently undergoing beta testing and will soon be released.

References

[1] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, Scott Shenker, “Controlling High Bandwidth Aggregates in the Network” Computer Communications Review 32:3, July 2002, pp.62–73.

[2] Cheng Jin, Haining Wang, Kang G. Shin, “Hop-Count Filtering: An Effective Defense Against Spoofed DDoS Traffic”, Proc. of the 10th ACM conference on Computer and communication security, 2003, pp.30–41.

[3] Daniel J. Bernstein, “SYN cookies”, http://cr.yp.to/syncookies.html, 1997.

[4] Haining Wang, Danlu Zhang, Kang G. Shin, “Detecting SYN Flooding Attacks”, Proceedings of IEEE InfoCom, 2002.

[5] John Ioannidis, Steven M. Bellovin, “Implementing Pushback: Router-Based Defense Against DDoS Attacks”, Network and Distributed System Security Symposium, February 2002.

[6] Livio Ricciulli, Patrick Lincoln, and Pankaj Kakkar, “TCP SYN Flooding Defense”, Comm. Networks and Dist. Systems Modeling and Simulation Conference (CNDS’ 99), 1999.

[7] Thomer M. Gil, Massimiliano Poletto, “MULTOPS: a data-structure for bandwidth attack detection”, Proc. of the 10th Usenix Security Symposium, August 2001.

[8] Gábor Fehér, András Korn, “RESPIRE – A novel approach to automatically blocking SYN flooding attacks”, Proceedings of Eunice 2004, pp.181–187.