Alert Flooding Attack on Snort and Its Mitigation

Introduction

Network Intrusion Systems employ a number of sensors for efficient reporting of attacks on the hosts in a network. But a serious problem with these sensors is that the information they produce is in a low level format and the system administrator gains no useful information from the report. In this report I am going to discuss about the usage of a method to correlate the alerts produced, the attack graph method. The subject matter of this report though will be about the discussion of a type of attack on NIDS called *'alert flooding'*. The first effect of this attack is the loss of service. The sensors do not do the intended job. The second effect is that we get lot of unintended alerts which make alert correlation impossible or at least makes correlation meaningless. The report will walk through the basics of NIDS, the operation of Snort, establishing alert flooding attack and solution to mitigate it using two concepts called *Token Bucket Filter and Queue Graph.*

How attacks are launched on NIDS

An NIDS is usually a software deployed on host in a network. An NIDS like Snort can be configured to monitor the network in a promiscuous mode such that it can examine those packets that are not intended to it. The NIDS detects malicious packets of data through signature based methods where possible attack packet contents are stored and if a match is found an alert is raised. The second method is the anomaly based one where legitimate traffic patterns are stored and any deviation is raised as an alert. In either of the cases there exists a communication channel between the NIDS host and the host of the system administrator. If this link is flooded with data the administrator will not receive the alerts in time and during which the attacker can send a malicious packet unnoticed by the system administrator. There are many methods to flood this channel but this report concentrates on one particular method of doing, using 'alert flooding'.

Core of the solution

There will be alerts generated by the IDS sensors. Alert correlation is done to match the alerts to exploits on the attack graph. Now the potential for attack exists when the attacker floods with fake attack packets which will make us construct an attack graph that gives false alerts and at the same time forces us to mix the critical alerts. So the solution analyzed in the report uses two components. The token bucket algorithm and the attack graph. The token bucket algorithm acts as the input for the attack graph. By using the attack graph we can eliminate alerts that are not useful in determining the attack scenarios. Using the token bucket we are able to defend against the alert flooding since it filters out such alerts through a mechanism described in forth comings pages. This implies that we will not miss upon crucial alerts due to alert flooding.

Basic operation of Snort

Snort works by comparing the incoming packets with the stored *signatures*. The signatures depict known attack patterns and are stored in a database. The contents of the packets on the network are scanned and compared with the stored signatures. If a match is found an alert is raised. Sample signature is shown below.

```
alert udp $EXTERNAL_NET any
->
$HOME_NET 31337
(msg:"BACKDOOR
BackOrifice access";
content: "|ce63 d1d2 16e7
13cf39a5 a586|";)
```

This signature is used to detect the 'BackOrifice' program generated traffic. The msg field is the message to be displayed and the content part of the signature is the data to be matched in the incoming packet. The protocol is UDP and can be generated by any host on the Internet and directed towards any node in the HOME_NET to port 31337.

What is Alert Flooding?

Alert flooding attack can be done by crafting a packet which matches any of the signatures and send large numbers of them towards the network. These packets may actually do no harm and are simply used to keep the IDS busy spending time logging spurious alerts. For example, using the sample signature shown in the previous paragraph, all the attacker needs to do is to craft a UDP packet whose content matches the signature given and direct it towards any host in HOME_NET to port 31337. It is important know the effects of this attack to appreciate its effectiveness.

1) The alert database becomes full and further alerts are not recorded. This may lead to missing true critical alerts.
2) The sensor reaches its threshold level and so its throughput is decreased. In the worst case sensor ceases to function and this leads to the total failure of the IDS.
3) The worst consequence is that the System Administrator is presented now with mixture of true and false alerts and so he cannot reason out clearly as to what could have caused the attack.

How easy is alert flooding nowadays?

There are tools available that automate alert flooding. Some of the popular tools are Stick and Snot. These tools use the freely available snort database. This database contains the signatures to of the most common attacks and are used by snort in all the networks wherever it's deployed. Using the signatures the tools craft the packets and flood them towards the network where snort will be operating.

There is also a severe vulnerability in the alerting procedure of snort. Researchers have recently found out that snort is flushing the buffer unnecessarily at two places which leads to unwanted system calls. This leads to poor performance in logging the alert. This flaw adds to the woes of alert flooding leading to quick drop of efficiency of snort.

Some of the methods to prevent alert flooding

There were attempts made by the developers of the snort themselves to address this alert flooding problem. The solution was to keep track of the *states* of all the TCP connections. As we know TCP has states and it's possible to track the state of any TCP connection. They redesigned Snort such that it doesn't accept packets from any connection that isn't properly established through the famous *three way handshake*. Now the attacker cannot send the packets as just 'single' packet but has to transmit three more packets in prior before sending his attack packet to. The primary requirement for establishing a handshake with a host is that they must already be communicating. This improvement is based on the report by Ptacek and Newsham.

They discuss this solution in detail and also enumerate the disadvantages in doing so. There is a very trivial disadvantage in this method. According to the design the IDS will record only those connections for which it has observed the 3-way handshake. This may lead to the IDS missing any TCP connection that had been established before the IDS started. Due to this loophole the attacker can evade detection without a 3WH.

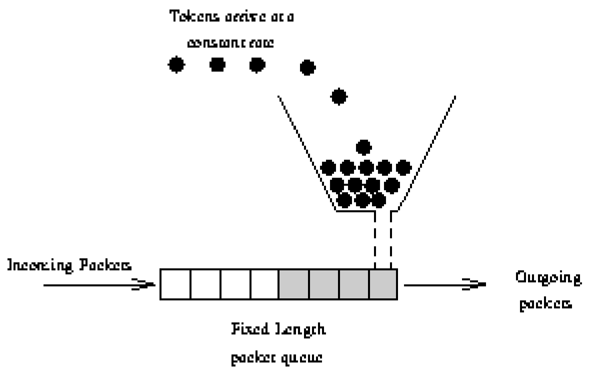
We define TCP Control Block as the set of data that the IDS needs to maintain to keep track of all the active TCP connections. A TCB is created by the IDS for every new connection and deleted once the connection terminates. There are four important parameters to keep track of called *source IP, source port, destination IP* and *destination port* for every active TCP connection and termed as *'connection parameters'*. For the IDS to reconstruct the information flowing through the TCP stream it should be able to figure out what sequence numbers are being used. The process of determining them is called *'synchronization'* and when the IDS is confused about this process it's said to be *'desynchronized'*.

By design, Snort uses the 3WH to determine the initial sequence numbers of the TCP connection. This is a blunder given the fact that it is possible for the attacker to produce a fake handshake. The attacker will be using the connection parameters which will be of some other connection will not be detected as the same parameters are being assigned to the attacker initiated connection. This effect is called *'desynchronization'*. This entire concept of keeping track of TCP states is implemented by Snort designers as '*stream4'*. But the cited disadvantages render stream4 useless for preventing alert flooding.

Another advantage for the attackers is that there are some exploits that can exist without any state information. The signatures for those exploits obviously do not care about the state of the connection. The attacker then uses those signatures to generate packets and flood them towards the IDS. To worsen the fact there are also some exploits that are stateless for which the *stream4* software will be of no use.

Token Bucket Algorithm

Token bucket has been a popular algorithm in the area of Networking used in reduce the flooding of packets caused by the source, by the destination and provide the required *Quality of Service*. We will now see how QOS can be deployed with Snort thereby mitigating alert flooding. There are two important parameters that token bucket employs. *Bucket size and token rate.*

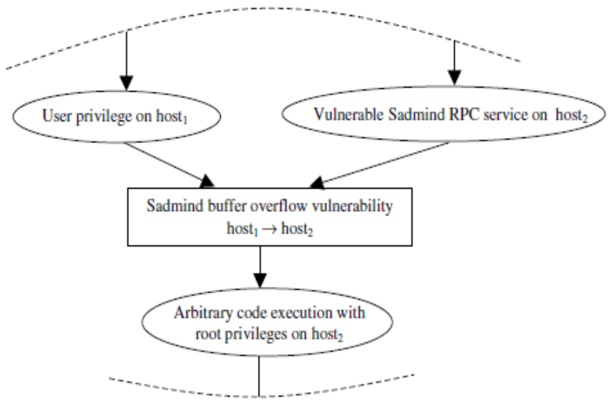


As we can see the tokens are generated at a constant rate '*token rate' and* are stored in the bucket. If the bucket becomes full the extra tokens are just discarded. Each arriving packet needs a token to pass through the filter. Packets that cannot obtain a valid token will be discarded. This condition is termed as '*over limit'*. If the *token rate* is greater than the *alert rate* it is termed as credit and can be used by alerts when the rate exceeds the normal rate, but for only for a temporary period.

Use of attack graphs in alert correlation

As I have been working in the Argus group research I am able to comprehend the difficulty in analyzing the low level result set produced by Snort. It is extremely difficult for anyone other than an expert Network Administrator to figure out what exactly caused the attack. There will be sequence of events that would have lead to the current compromised state of a host system. It is difficult to deduce that sequence by merely looking at the Snort produced result set which is *raw*.

Through the Argus research work I am also able to appreciate the usefulness of *attack graphs* in finding out that sequence of events. Let me just present a simple use of attack graph for a small event.


User privilege on $host_1$
Vulnerable Sadmind RPC service on $host_2$
Sadmind buffer overflow vulnerability $host_1 \rightarrow host_2$
Arbitrary code execution with root privileges on $host_2$

Correlation algorithms using attack graphs utilize the strategic relationship that exists between the alerts. To establish this relationship we need to map the alerts to vertices in the attack graph.

An attack graph gives knowledge about the vulnerabilities in the given network along with the information of how they are related. There are two types of vertices in an attack graph, *exploits* and *security conditions*. Exploits are vulnerabilities in the host systems. Security conditions denote the network status required or implied by the exploits. An edge from a security condition to an exploit says that the exploit cannot be executed unless the security condition is satisfied and an edge from exploit to security condition implies that execution of exploit will satisfy the security condition.

Referring to the sample attack graph *ovals* refer to the security conditions and the *rectangles* refer to the exploits. The graph says that the attacker can execute buffer overflow exploit only when the following two conditions are satisfied. The attacker should have access to the host and the vulnerable program must exist on the host. This is a simple attack graph giving us an obvious result but for a large enterprise network attack graphs provide some interesting attack patterns which are impossible to be detected just using the Snort alerts.

Correlation Methods

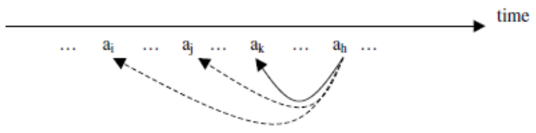
There are numerous correlation methods that are employed to match the alerts with the exploits in an attack graph. The first method is the *vulnerability centric approach.* This approach matches the *event type* attribute of alerts with the *vul* field of the exploits to find a matching exploit. This method can eliminate unwanted alerts like launching Windows-specific alerts on Unix machines. There is also a potential flaw. This method can miss those alerts that do not match any vulnerability but may prepare for other attacks, like **ICMP PING** that doesn't match any vulnerability will be probing for future attack.

There are other correlation methods that depend upon temporal characteristics of packets like order of arrivals and timestamps. This method may not be highly reliable as there are concerns regarding network delays as data may be collected from sensors located at different locations.

There is another approach called *nested loop approach.* The objective of this approach is to correlate alerts by searching exhaustively all the received alerts and finding out the alerts that prepare for a new alert. This may lead to high memory usage as there may be huge number of alerts to compare. An optimized approach uses a modified approach where we use a *window* approach where only those alerts that are nearer to the new alert are processed. This leads to a severe trade off between complete correlation and performance. Though it ensures good performance over the exhaustive approach chances are that we may miss crucial alerts outside the window. An attacker who knows that window technique is used can send his sequence of malicious packets delaying them such that they fall outside the window of their neighboring alerts.

Queue Graphs

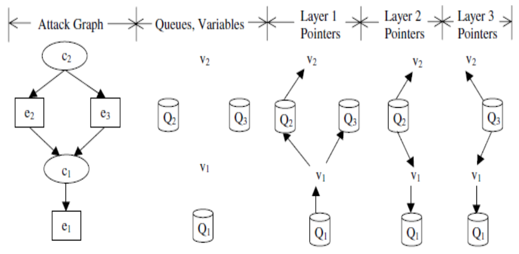
There is a new data structure called *queue graph* proposed by researchers that eliminates the drawbacks in the nested loop approach. The important observation made here is that the *correlation between alerts need not be explicit.*



The figure above shows alerts as they are received in the increasing time frame from left to write. Assume ai, aj and ak match the same exploit A on the attack graph and ah matches another exploit B but if A prepares B then ak and ah are implicitly correlated. There is also another implicit relation that since aj occurs before ak and ak occurs before ah A prepares for B aj must also occur before ah. This leads us to a new observation that a new alert needs to be correlated only with the latest alert in the past matching the same exploit. So we keep the correlation with other matching alerts than the recent one implicit. In this way the correlation algorithm needs to search backward to find the first matching alert unlike the nested loop approach.

Correlation using Queue Graphs

A Queue Graph is a in-memory data structure. It is designed as follows. Each exploit is denoted as a queue of length one. Each security condition is denoted as a variable.



The figure above shows the realization of queue graphs. The first is the picture of an attack graph. We need to construct a queue graph for this attack graph for in-memory correlation. For each exploit ei in the graph we need to do a BFS of the attack graph for each exploit. The third picture shows the BFS for exploit e1. For each edge connecting an exploit and a security condition we add a edge connecting corresponding queue and the variable. Here we connect to v2 using Q2 as either of them, Q2 or Q3 can be used to reach v2. Pictures 3 and 4 show the BFS done on Q2 and Q3.

Each incoming alert is matched with the exploit in the attack graph and placed in the corresponding queue. Since the queue is of length one, the queued alert is dequeued when a new alert matching the same exploit arrives. We need to correlate the alerts to derive attack patterns and for this purpose we use another data structure called *result graph.*

The procedure to record the alerts received and establish the correlation is as follows. First, for each new vertex received it is added as a vertex in the result graph. Next, a new alert matching the same alert will dequeue the old one matching the same exploit in the queue graph. An edge is added between these alerts to establish the temporal relationship between them. The new alert will have been enqueued in our queue graph. Once this is done we can just follow the pointers and search the queue graph. If an edge from a queue leads to another non-empty queue through a variable we record the relationship between in the result graph to aid in correlation.

Let me explain the correlation using queue graphs for the figure. Let ah match alert e1 and ai, aj and ak match e2. Their temporal order is shown in the figure. Alert ai is the first to arrive and it's placed in the queue Q2. After that aj arrives and forces Q2 to dequeue and adds itself to Q2. Now we need to update our result graph to record the temporal relationship between these two alerts. We do this by adding an edge between ai and aj. This also records the fact that ai and aj match the same exploit e2. Then ak arrives, dequeues aj. We record this fact as usual in the result graph. Finally ah arrives matching the exploit e1 which results in it being enqueued in Q1. Then we do a BFS search in the queue graph to establish the correlation.

As we can easily see a search from Q1 upwards leads us to non-empty queue Q2 thereby establishing the explicit correlation between ah and ak. This is exactly we wanted, establish correlation only between only one alert among a large number of alerts matching a single exploit. The queue graph approach is definitely an improvement over the nested loop approach which is vulnerable as it has the window concept where the correlated alerts may fall outside window evading correlation. But in queue graph once an alert matching an exploit is detected and queued, it can be dequeued only when an alert matching the same exploit arrives. In other words, after an alert, if the attacker tries to inject unrelated alerts to avoid correlation the queued alerts are not going to get removed which eases our correlation process.

Reducing data to aid in efficient correlation

This part of the report will discuss about using the throttling algorithm in combination with correlation. One of the researchers sees this problem of alert flooding from the point of view of the IDS operator who is a human being. This makes sense as the administrator is the one who has to make decisions in real time if any attacks are being launched. If alert flooding is done on the IDS obviously he will be presented with numerous alerts and will not be able to produce a valid attack trace.

The researchers propose two methods to solve this problem. One is to increase the resources at disposal and another is to decrease the amount of resources required. Increasing the resources means increasing the throughput of sensors used which will enable it to produce alerts constantly even if the packets are flooded at it. The obvious problem with this approach is that though we are able to scale the hardware to handle burst of data the human operator's abilities are limited. He still cannot make clear decisions out of these burst of data.

The second approach, of course the most promising one is to reduce the requirement of resources. This is done by using the token bucket filter algorithm which maintains the rate of incoming data. If the incoming rate increases the rate established by TBF (Token Bucket Filter) the packets are discarded. This suggests that when packets are flooded the TBF will gracefully discard those packets. But there is a potential problem with this approach that the algorithm runs the risk of dropping packets which are actually sent to attack the machines.

The solution proposed by the researchers uses the TBF along with the correlation algorithm in order to ensure that critical alerts are not missed. The exact approach is to add the TBF to each queue in the in the queue graph data structure we described. By this way flood of alerts matching an exploit will be discarded. There is an optimized method to inform the user about dropped alerts. The researchers use the Run Length Encoding Scheme (RLE). RLE is used to represent data that occur in large repetitions. The method RLE uses is, it writes the string followed by how many times the string has occurred. For example if ‘‘cat’’ occurs 1000 times then RLE represents it as cat 1000.

We can employ this technique into the queue graph data structure to inform the user about the excessive alerts that are dropped. Just add a counter to the queues in the graph and increment the counter every time an alert matching the same exploit occurs increment the queue counter. When the token bucket has enough credit so that we can dequeue, we remove the alert and its counter and make a log in our *result graph* data structure described earlier which is a permanent storage. An improvement is made to avoid discarding crucial alerts by applying TBF at the signature level. By doing this we are able to log even those alerts that do not correspond to any exploit in our attack graph. Fixing the parameters for the TBF depends upon the implementation environment but this task is trivial. For those alerts that do not match the exploit fixing TBF values may be quite challenging if not impossible.

Concluding Notes

Having realized the need for efficient correlation through the research discussions at Argus meet I did this study of popular methods existing till now to do efficient correlation of alerts in *real time*. Using queue graphs with TBF appears to me as a novel and a powerful solution to thwart such flooding attacks. The challenges in implementing this method lies in configuring TBF parameters and efficiently dealing with alerts that do not match with exploits in attack graph. But researchers point out that this can be done by the administrator after considerable experience.

References

[1] Wang et al, "Using attack graphs for correlating, hypothesizing and predicting intrusion alerts". *Science Direct, 2006*.

[2] Gianni Tedesco and Uwe AIckelin, "Data Reduction in Intrusion Alert Correlation".
The University of Nottingham.

[3] Thomas H. Ptacek, Timothy N. Newsham "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection". *Secure Networks, Inc, 1998.*

[4] Snort IDS,
`http://www.snort.org/`