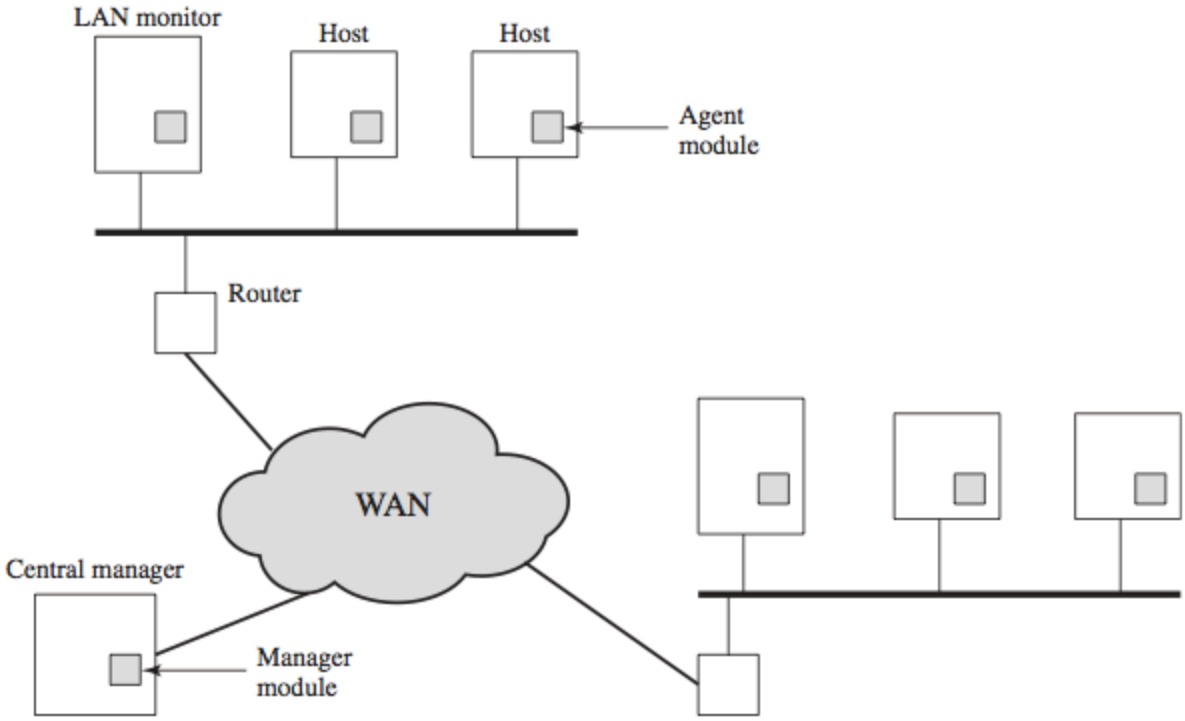
Distributed Intrusion Detection

by Philip Thyssen and Michael Franz

When looking at intrusion detection in the past the focus has been on single-system stand-alone facilities. A typical organisation, however needs to defend a collection of host distributed over a LAN or internetwork. Therefore there must be a better solution than to defend each host by stand-alone intrusion detection systems. This can be achieved by coordination and cooperation among intrusion detection systems across the network.
Porras however points out the following issues in the design of such a system:

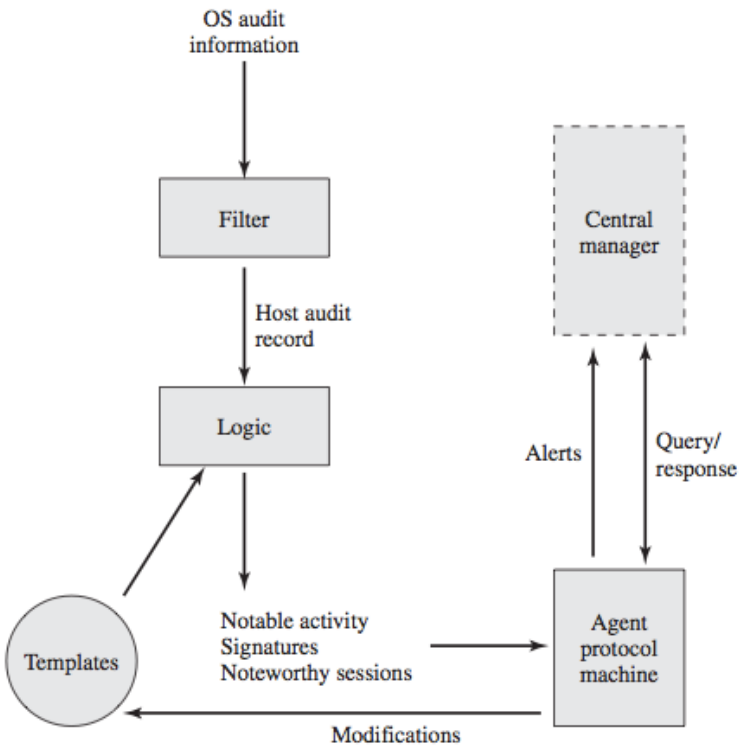
- The system should be able to handle different audit records from different systems.
- Audit records must be transferred securely over the network, ensuring the integrity and confidentiality of this data.
- Centralized or decentralized architecture can be used. One central point of collection or analysis or more if decentralized.

An architecture for Distributed Intrusion Detection could look like the one developed at the University of California at Davis [HEBE,92, SNAP91]:



- **Host agent module:** Background process for collection of security related events and reports to the central manager.
- **LAN monitor agent module:** Like the host agent, but collects data from LAN traffic and reports to the central manager.
- **Central manager module:** Receives reports from LAN monitor and host agents and correlates the reports to detect intrusion.

The above is designed to be independent of any operating system or system auditing implementation.



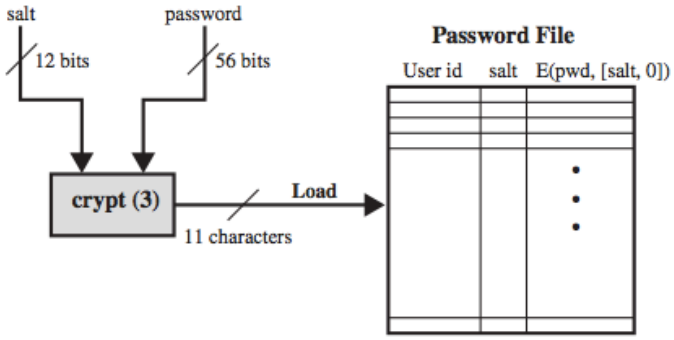
The figure above shows how the audit records are first filtered such that only security related events are collected. The records are then reformatted into a standard format referred to as the host audit record (HAR). Next the suspicious activity is analysed by a template-driven logic module. When suspicious activity is detected, an alert is sent to the central manager.The central manager can then draw inferences from received data, but may also query individual systems for HARs to correlate with HARs from others.
The system above is quite general and flexible. It is able to correlate activity from a number of sites and networks to detect suspicious activity that would otherwise remain undetected.

Honeypots

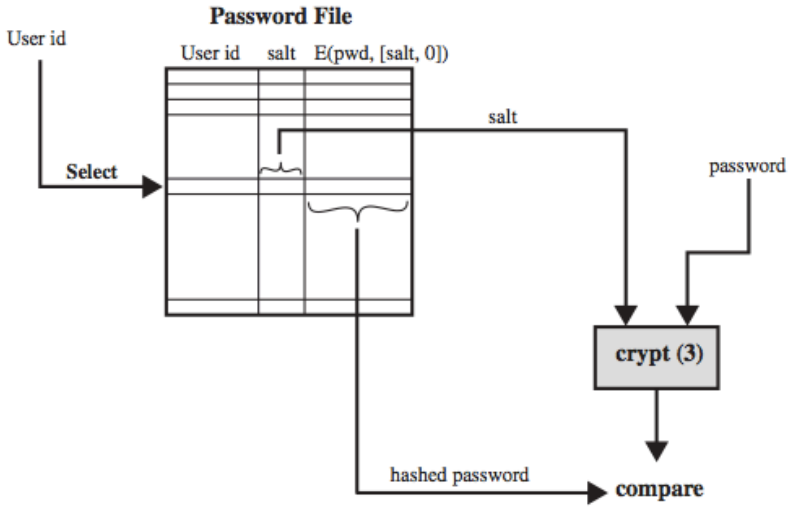
Another approach towards intrusion detection is using a system that acts and appears as a real system. The idea of Honeypots fits well with the name, lure an attacker into the system that such that he can be monitored and acted upon.
A honeypot is typically used when trying to divert an attacker from very critical systems. It only contains data that has the appearance of being valuable.
The strategy of the system is to make all attacks look successful and trying to keep the attacker on the system long enough, for an administrator to respond. A very complex monitoring and logging system is part of such a system, to collect as much data as possible. This data can be used to improve the real system, especially if new exploits are logged.
The first systems that implemented honeypots were simple systems typically consisting of only one computer. With attackers developing, honeypots have become evolved consisting of a structure that almost has the same complexity as the real system.

Password Protection

The front line of defense against intruders is of course the password system. Usually users have to provide not only a name or ID but also a password. Where the password serves as an authentication for the ID. The ID determines whether a user is allowed to gain access, and the privileges accorded to the user. The user might be a superuser who has access to more than a normal user.
Consider the following scheme which is widely used on UNIX, where passwords are never stored in the clear:



(a) Loading a new password



(b) Verifying a password

Each user selects a password of up to 8 characters, which is converted into a 56-bit value that serves as the key input to an encryption routine. Here the ***crypt(3)*** encryption routine is used. The routine is based on DES but is modified using a 12-bit “salt” value. This value is typically given based on the time of which the password was assigned. The hashed password is then stored together with a plaintext copy of the salt in a password file for the corresponding ID.
The salt serves three purposes:

- Duplicate passwords would not be visible in the password file, since they have different salt values.
- Increase the length of the password with 2 characters, without the user having to remember or know these.
- A hardware implementation of DES would ease the difficulty of a brute-force guessing attack, but is prevented because of 25 iterations of the DES algorithm.

When a user attempts to log on to a UNIX system using the assigned ID and password the operating system uses the ID to index the row of this user. Therefore ID's are unique. It then gets the plaintext salt copy and the encrypted password. It then encrypts the password provided by the user, using the salt which was just retrieved from the password file. If it matches then the password is accepted.
Instead of using a dumb brute-force technique(which is not feasible) of trying all possible combinations of characters, password crackers rely on the fact that some people choose easily guessable passwords.
A result of a study in Purdue University shows that some people choose absurdly short passwords:

Length	Number	Fraction of Total
1	55	.004
2	87	.006
3	212	.02
4	449	.03
5	1260	.09
6	3035	.22
7	2917	.21
8	5772	.42
Total	13787	1.0

Almost 3% of the passwords were three characters or less. Therefore a password cracker could easily try all possible combinations of 3 or less characters.
This could easily be solved by rejecting passwords less than a certain number of characters. But the length is only a part of the problem, since when people are allowed to choose their own password, they choose one that is rather guessable. This could be a name, streetname, a common dictionary word and such.
Therefore a password cracker could simply test the password file against a list of likely passwords(a crackers own dictionary of passwords).
A report about the effectiveness of guessing is reported in a rather old study [KLEI90]:

Type of Password	Search Size	Number of Matches	Percentage of Passwords Matched	Cost/Benefit Ratio[a]
User/account name	130	368	2.7%	2.830
Character sequences	866	22	0.2%	0.025
Numbers	427	9	0.1%	0.021
Chinese	392	56	0.4%	0.143
Place names	628	82	0.6%	0.131
Common names	2239	548	4.0%	0.245
Female names	4280	161	1.2%	0.038
Male names	2866	140	1.0%	0.049
Uncommon names	4955	130	0.9%	0.026
Myths & legends	1246	66	0.5%	0.053
Shakespearean	473	11	0.1%	0.023
Sports terms	238	32	0.2%	0.134
Science fiction	691	59	0.4%	0.085
Movies and actors	99	12	0.1%	0.121
Cartoons	92	9	0.1%	0.098
Famous people	290	55	0.4%	0.190
Phrases and patterns	933	253	1.8%	0.271
Surnames	33	9	0.1%	0.273
Biology	58	1	0.0%	0.017
System dictionary	19683	1027	7.4%	0.052
Machine names	9018	132	1.0%	0.015
Mnemonics	14	2	0.0%	0.143
King James bible	7525	83	0.6%	0.011
Miscellaneous words	3212	54	0.4%	0.017
Yiddish words	56	0	0.0%	0.000
Asteroids	2407	19	0.1%	0.007
TOTAL	62727	3340	24.2%	0.053

[a]Computed as the number of matches divided by the search size. The more words that needed to be tested for a match, the lower the cost/benefit ratio.

Here the following strategy was used:

1. Try the user's name, initials, and other personal information.
2. Try words from various dictionaries.
3. Try various permutations of the words from step 2.
4. Try various capitalization permutations of the words from step 2.

Here this resulted in approximately 3 million words. Using the fastest Thinking Machines implementation the time to encrypt these words for all possible salt values was under an hour. This could produce a success rate of 25% percent whereas maybe one hit could be enough to

gain access to the system.

Therefore, selecting a strong password(difficult to guess) is crucial for a system.

Password Selection

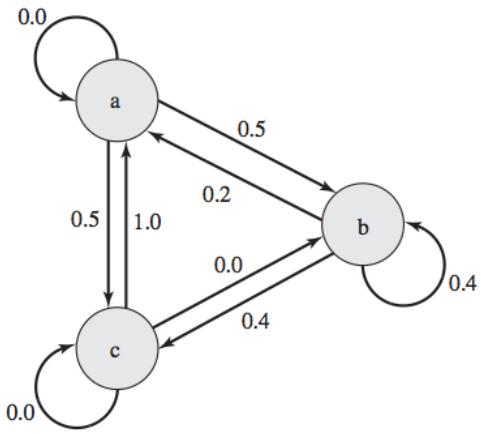
When looking at the previous section it is very obvious that the selection of the password is the most crucial part for ensuring security in a IT-system. When creating passwords it's very important to find a password that is not guessable, however it may not be so random that the user is not able to remember it. There are several techniques in use to in one way or another ensuring non-guessable passwords:

- User education: Enforcing rules on passwords such as minimum length, Capital and non-capital letters should be used and numbers and other characters should be represented. This method however can often only be used as a guideline to creating passwords.
- Generating random passwords: Will ensure that these are not guessable. The problem as mentioned earlier is that users will typically have difficulties remembering 8 completly random characters.
- Reactive password checking: If an active user's password is guessed by an dictionary attack, the user account is disabled. Weak passwords will be active in the system which yields a vulnerability.
- Proactive password checking: Before a password is accepted, it is tested on a Dictionary of guesses. Only if this test is passed, a password is accepted. Using a proactive checking technique is the most secure way of managing passwords. This ensures that only passwords are in use that would not be guessed, given an exhaustive dictionary.

This short introduction to all techniques showcases all typical technique used to avoid guessable passwords. Every technique is part of ensuring a good password selection. However, it's obvious that the proactive password technique is the most essential part. Therefore, most focus is given to this technique.

The most easy way to implement a proactive technique is to use one's imagination and fantasy to create a Dictionary containing all possible guesses. Such a dictionary would have to read linearly when a password is checked. Typical there are two drawbacks of this straight out implementation. First, the time it takes to verify a passwords can be very long depending on the dictionary, due to a linear search. Second, the size of such a dictionary can be large such that space can be an issue. These two points are made by the book, *Network Security Essentials*. One might doubt these two points in the light of recent hardware development. Nonetheless two techniques are presented which are based on dictionaries. Both techniques uses the data to create a structure which is more efficiently usable.

The first technique, uses a markov model to predict the probability of a password being guessable. To illustrate the concept, a simple example is used containing only three characters in its alphabet. The state of such a system is always the most recent letter, the transitions between different states is indicated by how likely such a state change is. This is showed in the figure below:



$$M = \{3, \{a, b, c\}, T, 1\} \quad \text{where}$$

$$T = \begin{bmatrix} 0.0 & 0.5 & 0.5 \\ 0.2 & 0.4 & 0.4 \\ 1.0 & 0.0 & 0.0 \end{bmatrix}$$

Looking at this model yields what components such a model consists of [*m*, *A,* **T**,*k*]. It's described by the number of states *m*, which states *A*, a transition matrix containing the probability of state changes **T**, and the order *k*. In this context, order means that the probability of a transition depends on the *k* previous letters. The model above is based on a simple one-order model.

If one wants to use " aacc" as a passphrase, it can be seen that the probability is 0 (0*0 *0.5*0*0). This means that the dictionary which the transition matrix represents did not contain any words which had "aa" and "cc" in them.

The interesting part is how to construct such a transition matrix. The book illustrates how to create a second order model based on a dictionary. This include three steps:

1. Constructing a frequency matrix **f**. An entry **f**(*i,j,k*) denotes the value of how often the characters *i,j,k* occurs right after each other in in all words in the dictionary.
2. Constructing another frequency matrix **g**. An entry **g**(*i,j*), is the total number of occurrences of three characters in a row that starts with *ij*.
3. Now the each entry for **T** can be calculated easily:
   T(*i,j,k*) = **f**(*i,j,k*)/ **g**(*i,j*)

This means that each entry is calculated as the number of occurrences of 3 characters coming right after each other divided by the total number of the two first letters occurring right after each other.

This means that such a model represents very well how a dictionary looks and can give a probability of how likely a password can be guessed. A weakness of this model is however that an attacker can anticipate such a model by building a markov model based on the attackers dictionary. It can be assumed that these dictionaries most probably will be alike, therefore an attacker can add words to it dictionary that the Markov model predict as very unlikely.

Another technique is based on a Bloom filter. A Bloom filter of order *k*, consists of *k* independent hash functions $H_1(x)$, $H_2(x)$,..., $H_k(x)$. Where each hash function maps a password into a hash value in the range from 0 to N - 1. Furthermore, the number of words in the dictionary is denoted by *D*. The hash functions and the dictionary are used in the following way:

1. A hash table of N bits is created with all bits initially being 0.

2. Each word in the dictionary is “hashed” through all hash functions. The corresponding hash table entries of the yielded values are set to 1.

This procedure draws a landscape of the dictionary in the form of a hash table. When a user proposes a new password it’s run through all the hash functions. If all the entires in the hash table corresponding to the values yielded by the hash functions are one, a password is rejected. This result would mean that a password most probably would be in the dictionary.
This technique however yields a problem of false positives:

$$H_1(\text{undertaker}) = 25 \qquad H_1(\text{hulkhogan}) = 83 \qquad H_1(\text{xG\%\#jj98}) = 665$$

$$H_2(\text{undertaker}) = 998 \qquad H_2(\text{hulkhogan}) = 665 \qquad H_2(\text{xG\%\#jj98}) = 998$$

Looking at the example above, it shows that in some cases a password that obviously would not be represented in a dictionary still yields a rejection. This is problematic if there are too many false positives. A short analysis yields that the probability of having false positives can be calculated in the following way:

$$P \approx \left(1 - e^{kD/N}\right)^k = \left(1 - e^{k/R}\right)^k$$

Where R is N/D. This is ratio hash table size to dictionary size. The idea is to rewrite this equation such that number of hash functions and the probability can be used as function parameters and the output is R, this would look like this:

$$R \approx \frac{-k}{\ln(1 - P^{1/k})}$$

This can now be used to determine the ratio R, given the number of hash functions and the wanted probability. The main purpose of analysing this part is to ensure a low probability of false positives while keeping the hash table as small as possible. The most important part of the Bloom filter approach is the ability to determine fast if a password is in the dictionary or not. This is possible due to the hash functions.