A Gentleman's Guide to Intrusion Detection and Protection

Mark Sitkowski

Design Simulation Systems Ltd

http://www.designsim.com.au

Abstract

Nobody ever broke into a bank's IT system by cracking a user's password. It's not cost-effective to waste computer time on such a pursuit, for the sake of the few thousand dollars that may, or may not be in the user's account.

It's far more cost-effective to persuade the bank to let you have access to its database, via a back door. Then, you have access to all of the bank's resources, for the expenditure of a minimum of effort, and without even having to understand how the authentication system works.

On the other side of fence, when your company's product actually is that bank's authentication system, and which it describes as 'Uncrackable', you have to expect this to be like a red rag to a bull, as far as the world's hackers are concerned.

Every day, dozens of them try to break the algorithm, but none ever succeed, so there is some excuse for the complacency which ensues. However, you soon notice that, for every front door attack, there are over a hundred attempts to totally bypass the authentication system, and get in via a back door.

Now, after you've told the world that the authentication system is uncrackable, it would be rather embarrassing to find that the hackers had decided not to bother cracking it, but had broken into your authentication server, instead, and hijacked your database.

You have no control over how the average bank, securities trading company or whoever uses your product, configures their online access server or ATM machine, but you can lead by example, and make sure that your authentication server, at least, can be made hack-proof.

Easy, right? All you need to do is to buy a device which will alert you, as soon as it detects a hack attempt, and prevent it succeeding.

If, after a few weeks of searching on the internet, and talking to prospective suppliers, you find that nothing on the market will do what you want, what do you do?

You write your own, of course...

Defining the problem

First, when you set up the infrastructure for your server's website, you need to do all the right things.

The only open port should be port 80, there must be no GET permission for cgi-bin, no POST permission for htdocs, all other methods like MOVE, DELETE, COPY etc need to be disabled, and there must be no interpreted CGI scripts, like those written in java, perl, shell or ruby.

A really good practice, is to try and ensure that the only HTML page is index.html, How do you do this? You dynamically create all of the other pages with a CGI executable - which, of course, must be a machine code executable, written in a compiled language. (In case you didn't realise it, this is done by including every line of HTML in a printf statement).

That way, if a hacker runs Wget on your site, he'll get no additional clues as to which page called which CGI, or what any of the HTML variables mean.

Bulletproof.

As far as it goes, it certainly is.

You'll probably get many connections each day, from the usual hopeful hackers, who will try to get in by breaking your authentication algorithm, and from the old-timers and incompetents, who will try buffer overflow, not having heard that that particular method didn't work on modern network applications.

Then, after a few months, things will probably change, as dozens of more determined hackers, with no life of their own, decide that they can combine distributed denial of service attacks with hack attempts. You'll probably be inundated with hundreds of queries, each designed to plant or exploit back doors, inject SQL or exploit vulnerabilities in every file whose name ended in '.php'.

Even if you don't use WordPress, cPanel, Joomla, ccmail or any of the other traditionally exploited software packages, and are immune to all of these attacks, it'll be extremely annoying to watch the server logs scrolling like a Las Vegas slot machine, as every unimaginative hack script repeats the same dumb vector, anything from two to four hundred times.

Also, it will be eating up your network bandwidth, and making the site respond less quickly than you would have liked, not to mention giving perverted pleasure to some hacker, watching hundreds of lines of hack script execute.

So, what do you do? You'll probably try manually adding a firewall rule to block the IP addresses of the worst offenders.

This, however, is time consuming and not very responsive, so it's obvious that something positive has to be done to stop this nonsense.

You'll probably decide to find an intrusion detection system which, everyone agrees, will solve your problem,

However, the first disappointment comes when you see what the market has to offer. Most of the IDS's are rule based. This means that they keep a list of all the known hacker sites, and block those IP addresses. Naturally, this list changes from day to day, so it needs to download a new list on a regular basis.

This is totally useless. You want something that's a lot smarter than that, and doesn't have to traverse a list of 20,000 IP addresses, each time anyone connects to your site.

You make a list of the functions you want your ideal IDS to perform.

First, it has to be content-based, so it can identify a hack attempt by the kind of thing the query is trying to do, which implies that such a system will need a certain amount of intelligence.

Second, having identified the hack, it will need to remember the IP address, drop the connection, and make sure that that IP address will never again be allowed to connect to your site.

Last, it would need to do all this in less than one second. The attacks that we face these days, are not directed from Mum and Dad's Wintel PC, but from high-end Unix servers in data centres.

You remember the speed at which your log monitor scrolled up the screen when you were under attack, and then examine the access log,. You notice that the average zombie hijacked server can shower you with hack vectors at a minimum rate of two or three a second and, sometimes, if they'd hacked a decent machine, up to ten a second.

Your goal is to stop it after the first vector.

The search for the product

What you expected was, that you would make a quick list of suitable products, then spend a long period of decision-making, choosing from many suitable candidates, until you found the ideal product. This will turn out to be another huge disappointment.

You notice from the first day, that the vast majority of intrusion detection systems are really no more than fancy java, shell and perl scripts, with a response time similar to that of a whale trying to turn itself around.

Disillusioned with the (not so) cheap end of the market, you decide that something used by banks has to be of the right quality, so you take a look at the professional, so-called 'enterprise level' products.

While researching this kind of product online, the whole thing gets off to an unpromising start, when you read the comments of a security consultant to a bank, describing the product they used.

During his speech, he declares proudly, that they could be aware of an intrusion within forty-eight hours of its happening.

Forty-eight hours? To you, forty-eight seconds would be too long, never mind forty-eight hours.

Predictably enough, the search of the high end of the market shows that shell scripts can also be available at high prices.

Worse, most of this stuff only runs on Windows, and you're a Sun Solaris and Linux shop. Who, in his right mind, would run a website on Windows?

During the demo of one of these products, the salesman explains that his system takes its data via a network connection to the actual web server machine, and it has this absolutely mind-blowing graphical display of how your website is being hacked, minute by minute. This is impressive, and a lot easier than watching lines of text scrolling up the screen.

You ask how it works, and are told that it counts the number of queries received in a given period and, if that exceeds a given value (which you could preset, of course) it would flash a lot of lights on the panel, and sound an important-sounding alarm bell.

Yes, but how did it differentiate between a legitimate connection, which just happened to be from a particularly fast machine, and a hack script? Well, it didn't, but the final decision would be up to its operator. Wait a minute, did that mean that it didn't automatically cut off the incoming connection? That's correct. The system administrator would have to do that.

The salesman explains, rather frostily, that what you actually wanted was an intrusion protection system, not an intrusion detection system.

Since his product is totally unaware of the content of each query, you reject it, and take a look at another, which claims to be content-aware. This is more promising, since it is possible to pre-program the thing with a selection from a set of internally stored, popular hack strings, and have it do the usual light flashing and frantic beeping when it discovers something interesting.

Although it runs on Linux, and a source code licence was available (at an additional cost), so that you can recompile it to run on Solaris, it, too, relies on the system administrator to do something about the hacker. Furthermore, there is no provision for adding new hack strings to the list hard-coded inside it.

Further questioning reveals that the thing runs like a packet sniffer, and reassembles each packet's payload to figure out the query string. This procedure results in many

false positives, and false negatives, and makes its response time less than breathtaking.

You are finally informed, that the only product which, apparently, does what you want is a proxy.

Filled with new enthusiasm, you take a cautious look at a few proxy offerings, only to be further disappointed.

Although a proxy really can do content-based filtering, the access by customers to your web pages through it proves to be virtually impossible. Also, the degree of remote control available is strictly limited, to the point of being unusable.

So, there it is. The market is willing to sell you a few Linux offerings, a huge number of Windows ornaments, but nothing that will examine the content of whatever is trying to get into your website, and automatically drop the connection, if it sees something it doesn't like.

The solution

During the time you were talking to the representatives of the various intrusion detection companies, you were thinking about the various issues which surrounded this problem.

Firstly, you absolutely need to know the content of each query. However, there is no way that you would accomplish this reliably, by tracking text strings across several hundred packets, and then reassembling the original query. The packet stream contains too much information, much of which is irrelevant, and identifying the query with any degree of certainty is too difficult.

What you need is a pre-assembled query, which is guaranteed to be a real query.

You are sitting in front of the apache access log monitor, in the middle of a botnet attack, watching it scroll enthusiastically up the screen, when it occurs to you, that what happens at the packet level is totally irrelevant. Bad things could only happen at the time when apache has the whole query in its buffer, and is about to act on it. Therefore, if you were to read the access log as it is being created, you would only be one line behind apache.

The hack queries themselves don't bother you, since they attempt to exploit vulnerabilities in software you don't use, so allowing one to slip through would be of no consequence.

So, the only criterion was to identify the first in a series of malicious queries from a given address, and do something before the hacker could send a second query.

Now the question arose, as to what constituted a malicious hack?

So, what is a hack vector?

You filter your access log, and remove all queries which access your legitimate web pages and CGI executables. What remains, according to Sherlock Holmes, has to be the truth.

A lengthy and detailed examination of the logs shows a rich selection of attempted hacks.

One that is extremely prolific, is a GET followed by a series of '../../..' of varying lengths, terminating in some significant filename, like /etc/passwd. This will have to be the first on your list, since so many hackers, most with no knowledge of Unix, think it has some chance of succeeding.

Next, you notice blocks of up to a thousand hexadecimal characters, each preceded by a percent sign. Decoding these, reveals that they are either IP addresses, or filenames, which some incompetent hacker assumes will slip past the casual observer. This hack's secondary function is an attempted buffer overflow, caused by its sheer length. A definite second choice for blocking.

Almost identical in purpose, is a similar hack, but with the percent sign replaced with '\x'. However, the hexadecimal values aren't ASCII. This is a puzzle, which takes a lot of research, until you recognise one of the hexadecimal values as being the Intel processor opcode for 'CALL subroutine'. Hackers call these things shellcodes, and the intent is to execute a buffer overflow, so they can place Intel machine code in the system's RAM, take over the CPU's program counter, index it to point to their own code, and execute anything they like on your machine. For any machine running on an Intel CPU, this would be the kiss of death.

With so many '\x' characters, this hack is easy to identify, so you add it to the list.

Then, there is the embedded question mark, usually followed by what looks like a script of some kind, and the embedded exclamation mark, usually in the middle of a lot of different hexadecimal stuff, which is obviously up to no good.

There are also the SQL injection hack attempts. I guess the most original, is one which attempts to overflow the CAPTCHA buffer (which you don't use) with a script like this:

"captcha/img.php?code=1'%20AND%201=0%20UNION%20SELECT%20SUBSTRING(CONCAT(login,0x3a,password),1,7)%20FROM%20User%20WHERE%20User_id="

You decide against wasting computing time on these, since your other criteria, such as the string '.php' and the percent signs, would easily identify it.

Finally, there are quite a few hacks containing an embedded series of plus signs, usually accompanied by a string of hexadecimal, or plain text like

‘Result:+no+post+sending+forms+are+found’. Just for the sake of complete coverage, you add a line of code to reject these.

Collating all of the information reveals something even more interesting. It becomes obvious that approximately ninety percent of all hack attempts of all kinds are aimed at dozens of different PHP files.

The attacks variy from simple GET queries and POST queries, to a pattern, where an initial query would attempt to GET a file like index.php (presumably, to establish its existence) and be followed by a second query, which would try to POST to the same file, and overwrite it with a back door. Then, a third query would try another GET.

In the light of these observations, you decide that another primary candidate for blocking would be any query containing the string ‘.php’.

Command and Control

What happens once the malware is installed on your computer?

Since Unix, unlike Windows, doesn’t permit self-executing executables, the hacker needs to access his malware after it has been installed.

How is he going to do so? Any self-respective server will have all ports closed except port 80, and believe itself to be totally impregnable. Unfortunately, this is not the case, since it is through port 80 that the C&C will wake up and direct the malware.

Almost all security devices concentrate on monitoring and defending TCP traffic through port 80. The C&C, on the other hand, talks to the malware using the UDP protocol, also through port 80, and is invisible to apache, and to many security systems.

It’s perfectly reasonable to block UDP traffic, with few resulting issues. However, just to complicate matters, there are other services, which run on UDP. DNS queries and replies, the Unix XDMCP login, and time server data are just a few examples. Any firewall rule which blocks UDP traffic, has to exclude these.

Dropping the connection

When you reach this point in the investigation, you can almost write the code for the content analyser in your head, and it is beginning to look more and more possible that you could write your own intrusion detection system. Then, you think about the tricky part. Dropping the connection.

The first thing to come to mind is a utility called tcpkill, which will very nicely drop an established TCP connection. However, a moment’s reflection shows that this would be inadequate. The average hack script re-sends the same line anything up to four hundred times and, if you invoked tcpkill every time, not only would the network traffic be no lighter, but the CPU would chase its tail trying to keep up with the

repeated hack attempts as well, especially when handling an attack from a few dozen servers simultaneously.

The next thought is, that you could use the firewall.

Since you expect that your IDS will be a stand-alone process, it would be necessary to use a firewall which was remotely programmable. Almost every supplier that you contact claims to have such a device, so things are looking very promising.

Unfortunately, firewalls are very security-conscious animals, and the only way to remotely program them, is to login to them first. The procedure for doing this is either through a gee-whiz graphical user interface, or via a telnet or SSH TCP connection. The GUI is obviously unacceptable, so you write piece of code which establishes a telnet connection to the firewall and send it a new rule. Ten seconds later, it is back on line.

Most firewalls contain a minimal Linux computer, and every time a new rule is added, this computer is rebooted. Even though ten seconds is a very short time for a reboot, it is just too long for your purposes.

Apart from the huge delay to add another rule, during that ten seconds, the machine would be sitting there with open arms, welcoming all hackers to do their worst, since the firewall would be resetting itself, and totally inoperative. Further, that ten seconds would allow several hundred new hack attempts to queue up for processing, resulting in a never-ending shuffle between our content analyser and the firewall.

Firewalls are abandoned, and you turn your attention to the Unix operating system.

Solaris has an extremely powerful utility, called ‘ipf’, which is a version of the ‘ipfilter’ module, which dates back to SunOS 4.1.3, in the good old BSD days.

It has all of the facilities available in stand-alone firewalls, such as NAT, but the filtering is actually performed in the Unix kernel, making it extremely efficient. It gets its rule set from a file, which is a minor drawback, but you decide to try it, anyway.

In the back of your mind, is the fact that Linux has a thing called ‘iptables’, which works in a similar way.

You write another piece of code- which appends a new firewall rule to the file, then tell ipf to re-read the file and restart. You run a few tests, and find that the time delay is almost immeasurable.

This is actually not that surprising. Since the filtering is done in the kernel, there is no actual ipf Unix process. When a user issues a command to re-read the configuration file, the kernel activates a ‘read’ system call, which is internal to itself, so there isn’t even a separate process to re-spawn. The only delay, is the time taken to execute the disk I/O – which is always a high priority task, since the kernel knows it takes a long time.

You decide against including any facility to count the number of queries in a given time interval. If the purpose is to identify a DDOS, then the hardware firewall can do so, and adequately cope with it. Also, doing this would mean repeatedly stopping other processing for the duration of that time interval. This could add an order of magnitude to the response time.

The complete system

You now have all of the building blocks for a complete intrusion detection – or, more accurately, intrusion protection system.

On startup, the IDS will read the ipf configuration file, and store all of the rules in an array of data structures. This will put the IDS in sync with the ipf packet filter, which is necessary, so that you don't try to add a rule for an IP address which is already being blocked.

Next, you would call a function which opens the apache access log, and performs a seek to the last line in the file. Having done that, it enters an endless loop, and waits for another line to be added to the file.

The loop contains the code of the content analyser, and has no time delays or pauses built in, so it will execute as fast as the CPU can execute machine code. This is usually extremely bad practice, since it uses 100% of the CPU's processing power. In your case, it doesn't matter, since your server has 32 CPU's, and devoting one of them to the IDS is a good investment.

As soon as apache logs another query, the content analyser scans it to see if it contains any of the hack signatures which you built into it. If a hack attempt is identified, a new firewall rule is automatically created, to make comparison with the stored firewall rules easier, then another function is called, to see if that IP address is already being blocked.

If this query turns out to be a new hack attempt, the ipf configuration file is opened, the new rule appended, and ipf re-invoked so it can re-read the file. Having performed the most important operations, the IDS then adds the new rule to its internal store.

There is a great temptation, when designing a system like this, to use multi-threading, or parallel processing. Although this would considerably speed up part of the processing cycle, the dangers of collision, between threads or processes, in areas such as file reading or writing is too great. Semaphores and mutexes are traditionally used to obviate such problems but, in general, if you need to use a mutex, you're either doing it wrong, or you shouldn't be multi-threading.

Conclusion

The system, as described, will give a number of false positives.

You will find that some query strings, especially those which are links from some online magazines, and some social media sites, contain elements containing the string '.php'. This is enough to trip the content analyser, and have the IP address blocked.

You may decide that you are willing to write this off as acceptable collateral damage, when compared with the enormous benefit of limiting each hacker to one hack attempt per lifetime.

However, just out of politeness to sites such as LinkedIn, and a few significant online publications, like Harvard Business Review, New York Times and others, you can add a few lines of code into the loop, and force it to ignore any positives containing the names of chosen sites.

The system works impeccably. It responds in under a second and, out of over a quarter of a million hack attempts, the parasites still haven't scored a single hit.

Too lazy to write the code? Have a Sun Solaris box? Send email to Mark Sitkowski at xmarks@exemail.com.au, and you may get the source code. If you run Linux, you'll need to rewrite the few lines of ipf code to do the same trick with iptables.

The hacks and exceptions are hard-coded and, if this were a commercial product, you would probably have it read a configuration file on startup. However, since it isn't, this is an exercise left to the reader. We don't mind recompiling the code each time a new hack query surfaces. It keeps it more secure.

Afterthought

It's very satisfying to have your IDS block all incoming hackers, and even more satisfying to watch the firewall logs, and see the parasites banging their heads against the firewall.

However, when they get tired of being unable to connect to your server, the same hackers are going to go away, and attack someone else. In fact, in a few weeks' time, you'll see the same IP address making another futile attempt at breaking in to your machine.

Wouldn't it be good, if we could get the hackers off the internet, as soon as we detect them?

The Enhancement

There's an urban myth, that hackers disguise their IP addresses, so you can't trace the hack back to them.

This is untrue. They hack servers in data centres around the world, plant malware on them, and let the malware run under remote control, infecting other servers with the same malware. Usually, the malware tries the next numerically adjacent address,

and looks for the same vulnerability. For the last few months, it's been Joomla, using a thing called 'BOT for JCE'

After an extremely short time, the hacker has a botnet, which he can sell to cyber-criminals, or use for his own ends.

What can we do, to minimise this damage?

Remember, we already have the hacker's IP address, embedded in the hack query, so we can use this, not necessarily to find the hacker, but to alert the owner of the address, that one of his servers has been compromised.

The first thing you'll need, is a whois database.

Ignoring the sharp intake of breath, let us proceed, armed with the knowledge that this requires only half a dozen tables, and a few megabytes of disk space. The trick is in the storage of the addresses, which is in numeric ranges, instead of individual machine addresses.

As soon as the IDS picks up a hack, it starts an SQL query on the whois database, extracts the owner of that address, together with a contact email address.

Anyone familiar with Unix can guess the next step. All Unix systems come with a thing called 'sendmail'.

Sendmail doesn't need an email account, it just needs an internet connection, and it is perfectly capable of delivering mail, if you feed it the information in the correct format.

Now, you're thinking that this will create a huge processing overhead, and slow down the response time of the IDS, to the incoming hacks.

Not so. Once we've identified the hack, we fork the IDS process, and call the function which does the whois lookup, then call the function which sends the email.

Well, not quite. The ISP will expect any report to contain log extracts, with IP address, query, date and time, so we first call a function which reads apache's access log, and extracts all entries containing the offending IP address. We assemble the log extract into a generic message, and pass this to sendmail.

All of this processing is being done in parallel to that done by the IDS, so there is no overhead. Best of all, the whole operation is totally hands-free, and your sys admin can sit back and watch hackers being removed from the internet.

This is a lot more positive than reporting the addresses to online lists, where only the owners of rule-based IDS will ever see them, while the hackers continue to operate with impunity. All ISP's hate hackers as much as we do, and are only too pleased to receive reports of compromised servers.

As we said earlier:

Too lazy to write the code? Have a Sun Solaris box? Know how to operate a compiler? Send email to Mark Sitkowski at xmarks@exemail.com.au, and you may get the source code of the enhanced IPS.

By the way, you can get a whois database from the internet, for about eighty dollars and, if you run Oracle, we can give you a set of migration scripts, to load the data. As a special favour, we'll also give you the abuse contact details of tens of thousands of ISP's in about 70 countries.