Intrusion Protection in Linux: The development and trial of a host and network intrusion protection system which satisfies the Jericho forum’s proposals for de-perimeterisation

HEMIS number 100548391

Supervisor: Dr. Allan Tomlinson

Submitted as part of the requirements for the award of the MSc in Information Security at Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from the published or unpublished works of other people. I declare that have also read the statements n plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences and in accordance with it I submit this project report as my own work.

Signature 100548391 Date

Table of Contents

Page

List of tables

Table of figures

Figure Page

Executive Summary

In this dissertation a hybrid host and network intrusion detection/protection system based on the Linux Netfilter framework was developed. The Network Intrusion Detection/Protection system is implemented as a kernel module and hence reduces the overheads associated with processing packets through libpcap as done in other IDS solutions.

The Host Intrusion Detection/Protection system is implemented as a hook to the standard C-library; it checks the integrity of binaries as they are executed by creating an MD5 hash of the binary and comparing this generated hash against an immutable stored hash database.

Keywords

Netfilter, Intrusion Detection, Intrusion Protection, IDS, IPS

Chapter 1.0 Introduction

The objective of this dissertation is the creation of an intrusion protection system which will protect the system on which it is installed from both network and system centric[a] attacks. We will be basing the development upon a commercial off the shelf (COTS) Linux distribution.

In [59] the mission critical mathematically provable correctness and existence of a formal specification that existed in the design of the software which controlled the Royal Navy's Merlin helicopters is contrasted with that which exists in a typical COTS system, Windows 2003.

The implication being that unlike the Royal Navy's Merlin helicopter software which was engineered for mission critical applications, was formally described and mathematically correct, statistically vulnerabilities are more likely to creep up within COTS systems. This due to their exponentially larger code bases which make formal analysis, mathematical and technical robustness extremely difficult to achieve.

When compounded with business pressures that motivate a penetrate and patch mentality within modern software development, COTS systems are fertile ground for the existence of vulnerabilities.

[a] System centric attacks have been called masquerading attacks in literature as the attacker has masqueraded as an authenticated user.

The debate between open source development and closed source development with regards to security has raged on for years; however it is widely accepted that open source software allows public scrutiny of the code base and therefore achieves security though openness and community development.

Public scrutiny of the code base and community development is itself vulnerable to people maliciously backdooring[b] the code base as happened when the Linux Debian distribution CVS systems were hacked and the Linux 2.6 kernel was backdoored [35].

Amongst this uncertainty about the security of the underlying COTS operating system either intrinsic due to the difficulty in retrofitting robustness into a system with already such a huge code base or through vulnerabilities which get introduced accidentally during development or maliciously we must implement controls which aim to mitigate the risks that inherently exist.

Such controls are not only a technical requirement, they have now been ingrained into various corporate governance requirements and information security related regulations.

[b] A backdoor is an intentional alteration to a program which would allow an intruder unauthorized access to the system at some point in the future.

For example, the Sarbanes Oxley Act of 2002 [1] of which section 404 requires senior management to establish and maintain an adequate internal control structure. Senior management must also produce a report which will include an assessment of the implemented controls and their respective effectiveness.

This boils down to information security terms as implementing systems and the management frameworks to support them. Such systems could take the form of intrusion detection/protection systems *i.e.,* controls, which are able to mitigate vulnerabilities.

Furthermore, today security researchers are publishing lots of vulnerabilities which are using web-based attack vectors. Their research has revealed particularly severe classes of vulnerabilities which developers were oblivious to until just recently.

Examples of such vulnerabilities according to [47] are input validation attacks such as SQL [49] and LDAP [20] injections, remote file inclusions [18], cross site scripting attacks[12], directory traversals and lately cross-site request forgeries [37].

The threat landscape therefore according to [50] is evolving away from one which was witnessed just a few years ago with attacks targeting network perimeters to one which at this time of age attackers are targeting web applications.

It can be argued that web-based attacks are more fruitful than attacks which try to penetrate hardened network perimeters. This is particularly true as most corporate firewalls are permeable to web-based traffic. Hence, web-based attacks go through to internal systems which attacks hitting at the perimeter would not have been able to access.

All this necessitates the need for a new approach to intrusion detection and protection, one that is more application level aware to counteract such emerging threats.

One of the main design objectives for the intrusion protection system which shall be further described in subsequent chapters is to be application-level aware. It achieves this through two modules, one, host based, which prevents unauthorized data alteration attempts on critical system binaries and one, network based, which monitors the network interface for network centric attacks by analyzing the OSI TCP/IP application layer.

Finally this concludes the introduction which described the motivations and need for intrusion detection and protection systems and more specifically those which are application-level aware. The next chapter will describe the organisation of this document.

Chapter 2.0 Document organisation

In the next chapter we shall be describing the current state of the art in intrusion detection and protection. The subsequent chapter shall describe desirable characteristics of intrusion protection systems. We will then discuss the implementation and how it matches these desirable characteristics.

We will then delve into the technical details of both the network intrusion protection system and host intrusion protection system. We will develop a typical attack scenario to test the effectiveness of the network and host intrusion protection systems.

Finally, we will describe how the implemented intrusion protection systems conforms to the de-perimeterisation ideal set forth by the Open Group’s Jericho Forum.

Chapter 3.0 Intrusion detection and protection systems

Intrusion detection systems (IDS) monitor IT systems for signs of intrusion by monitoring different aspects of the system for activity deviating from the norm which could be in violation of the security policy.

In [5] Stefan Axelsson gives a taxonomy of the current state of the art in IDS. He groups IDS in three broad categories; the first is anomaly detection. The second; signature-based and finally the third are called signature inspired IDS.

Anomaly	Self-learning	Non time series	Rule modelling
			Descriptive statistics
		Time series	ANN
	Programmed	Descriptive stat	Simple stat
			Simple rule-based
			Threshold
		Default deny	State series modelling
Signature	Programmed	State-modelling	State-transition
			Petri-net
		Expert-system	▨
		String-matching	▨
		Simple rule-based	▨
Signature inspired	Self-learning	Automatic feature set	▨

Figure 1 From [3] describing intrusion detection taxonomy.

Anomaly detection based IDS do not require prior knowledge of adversary behavioural patterns; they are able to distinguish which behaviour constitutes normal *i.e.*, baseline and which anomalous.

Anomaly detection systems can either be self-learning *i.e.,* IDS which are able to learn independently which behaviour constitutes normal and which anomalous. Alternatively the learning can be programmed as a set of rules which dictate which behaviour patterns are normal and which anomalous.

The methods through which these IDS achieve self-learning can be broken down according to [3] into non-time series of which rule-modelling, descriptive statistics are part and time series methods of which artificial neural network systems are part.

Non-time series self-learning is achieved through modelling of the normal behaviour of the system without taking time series behaviour into consideration.

An example of non-time series self-learning is rule-modelling which involves studying the traffic and subsequent generation of signatures that portray normal behaviour. When the observed traffic deviates from the generated signatures, the traffic is considered anomalous.

Descriptive statistics are another method of non-time series learning in anomaly based IDS. A statistical mono-modal profile is created by averaging previous observations using equation (1).

If the observed user behaviour or traffic deviates a certain number of standard deviations according to equation (2) then the behaviour or traffic is considered anomalous.

$$\mu = \frac{1}{n}\sum_{i=n}^{n} X_i \tag{1}$$

$$\sigma_{\mathrm{x}} = \sqrt{\frac{1}{n}\left\{\sum_{i=1}^{n} X_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} X_{\mathrm{i}}\right)^2\right\}} \tag{2}$$

Time-series self-learning IDS use artificial neural networks (ANN) to form a weighted non-linear statistical model over time of what constitutes normal behaviour.

An example of such a system is given in [46] and shown schematically in Figure 2 whereby the past commands issued in a terminal are fed into the neural network during the back propagation training phase.

After training, the ANN is able to predict what the next command will be based on what is currently being typed in the console. If the next command does not match, .then this is considered an anomalous event.

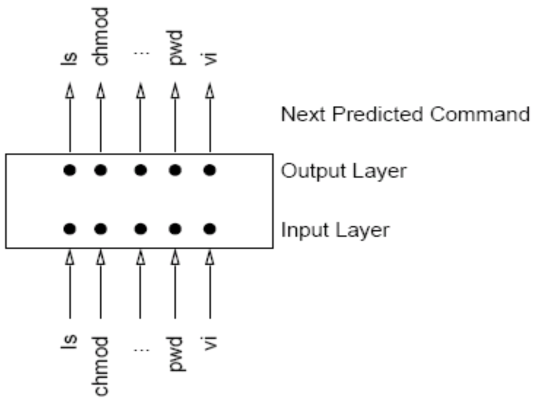


Figure 2 From [46] describing an ANN which is able to detect what the next command is going to be based on previous observed command typed into a console.

Anomaly detection systems which are not self-learning fall within the area of programmed anomaly detection systems. In programmed anomaly detection systems the learning is programmed as a set of rules which dictate which behaviour patterns are normal and which anomalous.

Within programmed anomaly detection systems, as described in Figure 1 above from [3] the anomaly detection profiles can be created using descriptive statistics as described previously or default deny policies.

Default deny policies, first create rules which describe all user-system interactions which are allowed by the security policy. Everything else is considered anomalous and counter to the policy, hence detected as a possible intrusion.

In state series modelling a set of secure system states are defined. Every interaction with the system causes it to evaluate whether the system should transition into the next secure state. If the transition required counteracts the security policy, then it is deemed as insecure and hence this state transition is considered anomalous.

The next class of IDS are called signature based or misuse-based IDS. Here, adversary behaviour is known beforehand and programmed into the detection routines of the IDS as signatures.

According to [3] such systems can be classed as default-permit IDS. This because, all attacks that are not explicitly listed as signatures fall through within the domain of allowed operations.

Programmed signature based IDS as can be seen in Figure 1, can be categorised into state-modelling, expert system, string matching and simple rule based IDS.

State modelling describes an intrusion as a number of prerequisite events which must all be present for the intrusion to have occurred. An example of this could be *e.g.,* if the Ethernet interface is in promiscuous mode[c] and we detect the binary Ettercap[d] running in the process listing, then somebody is sniffing on the LAN.

According again to [3] state-modelling can be further broken down into state transition models and Petri nets. State transition models can be imagined as a chain of events which must be traversed from beginning to end for the intrusion to be considered to have occurred.

---

[c] An Ethernet interface in promiscuous mode listens to all traffic passing through it rather than that solely destined for the machine the Ethernet card is installed in.

[d] Ettercap is a network sniffing program which sets the Ethernet card in promiscuous mode it can be found at http://ettercap.sourceforge.net/

In Petri nets as in state-models the intrusion traverses a chain of events however with the sole difference that the Petri net is able to pick up the intrusion at any point within the chain of events [3].

The next category of signature based IDS use expert systems to detect intrusions. An expert system according to [23] is a computer program containing a knowledge base and a set of algorithms and rules used to infer new facts from knowledge and incoming data.

In the case of an IDS the expert system knowledge base would contain profiles of what typical attacks look like *i.e.*, sequences of commands which when typed into a console would constitute a intrusion attempt.

One subsystem of the expert system would monitor traffic either host or network based and pass this traffic through algorithms which would classify whether what is being typed falls within one of these typical attack scenarios and if so, an alarm would be raised.

Very often according to [23] expert systems are implemented as rule-based systems which employ if-then-else scenarios. These if-then-else scenarios query the knowledge database at every stage categorising the threat according to the decisions made through the logic units.

String matching IDS use string matching algorithms which query a database of often case sensitive attack signatures and match these with observed traffic either host or network based. If a match is found within the traffic, then an alarm is raised denoting that an intrusion attempt has been made.

Finally signature inspired IDS systems are able to use the best of both worlds, anomaly and signature based IDS detection techniques to create a multifaceted view of the current normal or abnormal state of the system being monitored.

IDS's can be classed as either real-time systems or systems which process batch data [56] from *e.g.*, tcpdump[e] output. Real-time systems are naturally preferable in a real-world environment as they allow for response to events as they occur.

IDS systems whose source of audit data either comes from the host or the network are called host or network intrusion detection systems *i.e.*, HIDS, NIDS respectively [8]. The difference between the two depends on the data which is being processed.

[e] Tcpdump is a program able to print out packet headers coming from a network interface http://www.tcpdump.org

In the case of the HIDS the data being processed is critical system information such as system log files, binary secure state information *e.g.*, MD5 checksums, access to resources beyond the users allowed capabilities set within the security policy.

In the case of NIDS, the data being processed is packet information usually retrieved by setting the network interface into promiscuous mode. By looking at the packet headers the NIDS is able to make an informed decision whether or not the packet contains a malicious payload or is a subset of a larger number of malicious packets *e.g.*, a (D) DoS[f] attack.

A NIDS by itself can not monitor events that could constitute breaches of security policy happening on the computer, only those coming in through the network. Therefore, if both types of monitoring are required both NIDS and HIDS are installed.

Furthermore the response taken towards a given intrusion can be classified as either passive or active response; passive response simply notifies the administration staff of the event which occurred whereas active response takes defensive (and sometimes offensive) action to mitigate the risk without requiring human intervention.

[f] We have signified a DoS attack as a (D) to denote the many times distributed nature of such attacks.

The response taken towards an intrusion is the very fact that differentiates between an intrusion detection system, IDS, and an intrusion protection system, IPS. A slight play of words to the casual observer however technically an IDS is able to only detect attacks leaving the actual mitigative action to the operators judgement. As opposed to an IPS which takes initiative in dealing with the incident as described previously.

The problem which arises with anomaly detection is when conceptual drift occurs [58] whereby users learn to operate in new ways, new tasks are added and the system must be able to cope with the new normal behaviour baseline without raising false alarms *i.e.*, the system must be able to change as its legitimate users knowledge or job requirements evolve and they have to interact with the system in different ways.

The evolving characteristic can also be taken advantage of by attackers by pacing out their intrusion over a large timeframe therefore causing the profile to shift so as to match as normal behaviour their anomalous actions [58].

In this case, provided the signature database is up-to-date, it is signature-based IDS's which offer greater specificity. The downside being that the attack patterns must be known beforehand which has consequences with regards to detecting zero day attacks *i.e.*, attacks which are so new that as of yet do not have a detectable signature for them. In such cases, anomaly detection IDS excel as they are able to detect novel zero day attacks.

Now that we have described the various forms of IDS and IPS systems, the next chapter shall discuss the desirable characteristics that any given IDS or IPS system must possess.

Chapter 4.0 Desirable intrusion protection system characteristics

In [33] a set of requirements are implicitly laid out for IDS and consequently IPS systems. These requirements dictate the effectiveness of any given installation and system as well as its manageability and scalability.

The first feature which would be desirable within an IPS system is that of continuous operation. Operating continuously means that in any given moment in time the IPS is able to detect and protect the system against any impeding threats.

The system itself must be fault tolerant. This is a very broad term and in [2] it is defined as a system which is capable of self-diagnosis, repair, and reconstitution, while continuing to provide service to legitimate clients (with possible degradation) in the presence of intrusions or network segregation [34].

Furthermore the monolithic or modular nature of an IDS/IPS system plays a decisive role in the level of fault tolerance it is able to achieve. This is described in [33] where it is argued that monolithic IDS/IPS systems *i.e.,* systems in which detection, characterisation and finally mitigative action occur within one program pose a single point of failure and hence also are likely targets for attackers to try and disable.

Hence in [33] to achieve fault tolerant operation a set of distributed autonomous IDS/IPS systems are installed which utilise peer-to-peer type communication to notify each other of suspicious events and when these events exceed a certain threshold, the operator is notified.

Within the category of fault tolerance another desirable characteristic for an IDS/IPS system is that of tamper protection; hence making subversion costly in terms of time, effort and knowledge for the attacker.

Another desirable characteristic for any given IDS/IPS system is that of possessing a low overhead and therefore not impeding other parallel operations that might be performing on the systems on which the IDS/IPS is installed.

According to [34] enterprise networks are migrating from Fast Ethernet to Gigabit Ethernet technology at the core of the network. This subsequently means that heavy processing duty is placed on any given IDS/IPS system which must inspect each packet's payload for malicious content.

Inspecting traffic and application layer payloads at line speed can prove an impossible task. Therefore according to [34] to adequately protect a given system or the corporate network behind the IPS, the IPS must be capable of not dropping packets in even the most bursty conditions.

Another desirable characteristic for an IDS/IPS system implementation is that it must be easy to deploy. This boils down to creating intuitive interfaces, documentation and possibly even creating sensors for multiple architectures *e.g.*, a sensor for Linux systems, a sensor for Sun Solaris systems and a sensor for Microsoft Windows systems.

In [34] ease of deployability of an IDS/IPS system is also defined as one that possesses web-based management consoles with all the associated role-based access control. A web-based management console enables the IDS/IPS analyst staff to monitor the deployed sensors throughout the corporate network without requiring physical access to the device.

In [33] a requirement is that an IDS/IPS system must be adaptable and modifiable *i.e.*, addition of new rules, new sensors, learning new user and system behaviour.

This is particularly important as new attacks surface on almost a day-to-day basis. In [33] the distributed agent approach is heralded as not requiring complete rebuilds of the IDS/IPS system should a new attack surface and signatures need to be included within the build. Rather, just the subsystem responsible for that specific attack gets rebuilt. This aids in easier management, and faster uptime between scheduled maintenance.

Furthermore, an IDS/IPS system must ideally detect a broad range of attacks both known and unknown. By utilising a single device for network and host attacks and DoS mitigation enables the best correlation of attacks and therefore increases accuracy whilst also reducing procurement, installation and hence total cost of ownership [34].

This concludes the description of desirable intrusion protection system characteristics. We shall now describe our IPS implementation and refer to these requirements in our description of the implementation which is given in the next and subsequent chapters.

The IPS system as a whole is shown in Figure 3. It consists of two major subsystems; that of the host intrusion protector and the network intrusion protector, HIPS and NIPS respectively.

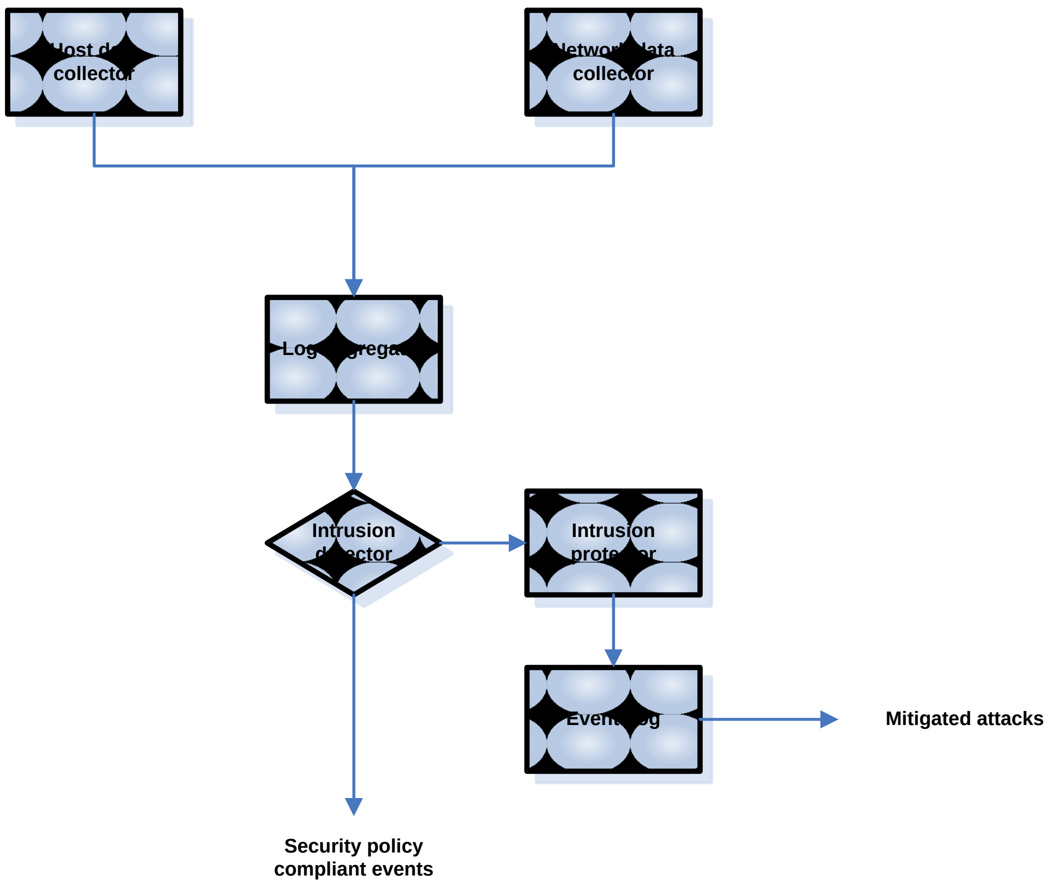


Figure 3 Describing the hybrid host and network intrusion protection system.

In [19] Spafford describes the file integrity checking program, Tripwire. In 1994 the motives for the existence of such a tool were to protect system databases and files which could be altered by an attacker either for purposes of denying entry to legitimate system owners, allowing him future entry on the system or masquerading his presence on the system through deletion of system log files.

To this day, little has changed with regards to a typical attacker’s modus operandi [36]. This is equivalently reflected in the success and market penetration of the company, Tripwire Inc which was based on the work in [19].

However as Tripwire was not explicitly marketed as a fully fledged IDS it falls short on our set of desirable IDS/IPS characteristics described in chapter 4.0. Namely the desirable characteristics of continuous operation, fault tolerance and system overhead.

Starting with the desirable characteristic of continuous operation, the approach used by Tripwire as described in [19] is that of periodically checking the integrity of binaries as a scheduled event.

This poses a danger as a window of opportunity exists between system scans in which an attacker might have maliciously overwritten a system binary which if executed by a privileged user might lead to a loss of confidentiality, integrity or availability of the system or even elevation of privileges for the attacker.

Our implementation of the HIPS performs binary integrity checking on runtime. Therefore at any given point in time a reasonable assurance can be gained that the currently executing program's integrity is assured.

Secondly the desirable characteristic of fault tolerance. Very much like Tripwire, the HIPS will perform binary integrity checking on critical system binaries. Both solutions use the MD5 message digest algorithm described in [42].

According to [19], MD5 was chosen over CRC as in 1994 it was found that CRC offered little collision resistance *i.e.,* a 32-bit CRC collision was able to be found in approximately four hours.

Since then, MD5 has been found to be equally susceptible to collision attacks whereby the effort required to find a collision is in the order of 2^{39}, as opposed to 2^{69} required according to the birthday paradox [60]. However, for most practical purposes MD5 is still very much used today, and will be used in the HIPS as well.

However within the topic of fault tolerance, it is the monolithic nature of Tripwire which is its weak point making it very much susceptible to disabling the system altogether. This can be done by *e.g.,* modifying the Tripwire binary itself or using the –*-update* command to update the signature database subsequently to modifying the binary in question so as to avoid alteration.

Although the HIPS can be disabled as well, as will be discussed in the implementation chapter, the IPS system as a whole is modular therefore the intruder will most likely be detected by the NIPS at the earlier stages of his intrusion attempt. This argument is supported in [33].

Tripwire does not stop a legitimate user executing a binary which as been maliciously or even accidentally altered, it merely detects that such an action has occurred. The HIPS is able to actively protect the legitimate user from running potentially harmful applications.

We will also describe the concept of an intrusion remediation system which is able to reconstitute the affected binary to its prior verified state. The HIPS therefore can be considered more of a fault tolerant system than Tripwire is.

Furthermore one of our desirable IDS/IPS characteristics is that of possessing a low overhead on the system. The HIPS performs its hashing and comparison operations every time a binary is executed, Tripwire on the other hand performs its hashing and comparison operations periodically.

Tripwire therefore creates bursty CPU demands during whole system scans. These might interfere with critical operations happening at that given point of time. Whereas the CPU demands required by the HIPS are evenly spread depending on use of system binaries.

To quantify the overhead associated with Tripwire integrity checking on a 20Gb file system, an experiment was conducted on a 3.4 GHz Intel Pentium computer with 512 mb ram, running GNU Debian Linux on a 2.6.15 kernel. Tripwire was invoked with the *–check* command.

The results are given in table 1.0. It can be seen that during integrity checking the average CPU load was 33.67%.

	CPU %
Measurement	Integrity checking
1.00	36.50
2.00	36.30
3.00	35.00
4.00	34.00
5.00	33.00
6.00	34.10
7.00	33.10
8.00	32.80
9.00	31.80
10.00	32.50
11.00	31.30
Average CPU %	33.67

Table 1 Experiment conducted on a 3.4 GHz Intel Pentium computer running GNU Debian Linux to quantify the overload associated with integrity checking a 20Gb hard drive.

This value is not exactly comparable with the overhead associated with that of run-time integrity verification as done in our HIPS. This because, the CPU load during HIPS run-time verification would also take into account the actual execution overhead of the program.

However in [5] and [54] user-level library interposition based IDS sustained a 3.2% overhead. Library interposition is the same method we will be using in the design of our HIPS.

The HIPS will operate within the confines of userland by replacing the standard C library call *execve* which is called every time a program is executed. Within our new *execve* call, we shall create a MD5 hash of the program to be executed and compare this with the MD5 hash in our database. If the two match then the binary has not been tampered with. Otherwise, permission is denied.

In [57] any given IDS/IPS device is characterised as a mission critical system that requires good detection coverage, economy in resource usage and resilience to stress. These requirements are very much akin to those set forth in Chapter 4.0 as our desirable intrusion detection/protection characteristics.

The current market de-facto NIDS is Sourcefire's Snort initially described in [44]. From its humble open-source beginnings it has as of March 2007 become an $86.3 Million corporation catering to small, medium and very large organisations.

Snort is based on the popular packet filtering library, libpcap [55]. Much like our NIPS it performs packet payload inspection. This goes to say that both our NIPS and Snort disassemble incoming packets and match the OSI application layer data with known attack signatures.

Snort according to [44] was never meant for high-speed networks. This is reflected in its use of the libpcap packet processing library. Libpcap is a user-level interface for packet capture.

Speed is of utmost importance in intrusion detection, this because networks can operate on multiples of Gigabits per second with speeds reaching even 10 Gigabits per second, inspecting traffic and application layer payloads at line speed can prove an impossible task.

If we recall from our set of desirable IDS/IPS characteristics in Chapter 4.0, one of our desirable characteristics is that of being capable of not dropping packets in even the burstiest conditions.

According to [21] intercepting packets in userland *i.e.,* using a packet filtering library such as libpcap, means that most packets are copied into userland from the kernel *i.e.,* a context switch occurs, only to be then discarded by the NIDS. Therefore, such solutions are slower at filtering packets than kernel-based solutions.

At speeds of one Gigabit per second, libpcap approaches have been found to function relatively well with not that many missed packets. However when surpassing this speed, specialized hardware based on network processors was required or distributed sensors which were load-balanced between them [41].

To further support this argument, in [45], kernel space filtering versus user-land filtering performance is compared on a Gigabit Ethernet network. The main conclusion being that when no additional processing is required to be done on intercepted packets, then it is user-level filtering that excels.

However, in NIPS/NIDS solutions substantial processing is applied on intercepted packets. The packets must be matched with the rule database to see if an impeding attack is in process. In [45], when additional processing was done on each packet (10 μs worth), it was kernel-based filters that excelled.

[45] goes on to say that if the emulator *i.e.,* the NIDS/NIPS, is used on a shared Ethernet segment or on a host receiving substantial amounts of additional traffic destined for other processes on the same network interface card then kernel level filtering excels once more. These are typical scenarios encountered in corporate, every-day networks.

The NIPS shall be intercepting packets within the kernel and hence will possess all the performance advantages gained in avoiding the context-switch to userland that libpcap approaches utilise.

The NIPS will operate by using the hooks provided by the Linux Netfilter Framework within the Linux kernel. By hooking into the Linux protocol stack at such a low level we have access to all seven layers of the OSI TCP/IP stack.

According to [32] a hook is a programming method which uses handler functions (*hooks*) which modify the flow of execution. A new hook registers its address as the location for a specific function, so that when the function is called, the hook is executed instead.

Although the proposed system as a whole is presented in the proof of concept stage, it is envisioned that with further development it will be able to provide a comprehensive defense in-depth solution for Linux based systems.

"*The defender needs to plan for everything… the attacker needs just to hit one weak point.*" King Darius *vs.* Alexander Magnus, at Gaugamela (331 B.C).

The next chapters shall describe the network intrusion protection and host intrusion protection systems in further detail

Chapter 6.0 Implementation

6.1 Core system

6.1.1 Network intrusion protection system

According to [7] and [41] the Netfilter framework is comprised of a series of Linux kernel hooks in various points in the protocol stack. These are given in table 1.0 below:

NF_IP_PRE_ROUTING
NF_IP_LOCAL_IN
NF_IP_FORWARD
NF_IP_LOCAL_OUT
NF_IP_POST_ROUTING

Table 2 Netfilter Framework Linux kernel protocol stack hooks.

According to [41] each and every one of these hooks can be called from within a loadable kernel module and hence intercept the packet at any given stage in its transit. These hooks are given a priority with regards to how ‘urgent’ their interception is *i.e.,* if given NF_IP_PRI_FIRST the packets are intercepted immediately.

Once a packet has been received by the network interface and determined not to be corrupt *i.e.,* CRC checksum computed then it is sent to the NF_IP_PRE_ROUTING hook. This hook determines whether the incoming packet is meant for the host or if it should be passed on.

Should it be destined for the local machine then the packet is passed to the NF_IP_LOCAL_IN which sends the packet to the application. Otherwise, the packet is sent to the NF_IP FORWRAD hook. NF_IP_LOCAL_OUT is called for packets leaving the host.

In any case, all packets pass through the NF_IP_POST_ROUTING hook whether they are destined for the local host or not. The above is also shown pictorially in Figure 4.

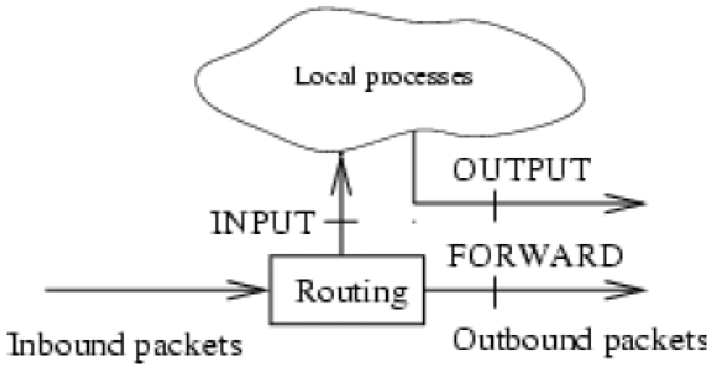


Figure 4 From [22] describing Netfilter module routing.

According to [7] and [41] once a packet has been received by the hook and packet processing is complete one of the following return codes are returned:

NF_DROP
NF_ACCEPT
NF_STOLEN
NF_QUEUE
NF_REPEAT

Table 3 Netfilter return codes.

NF_DROP signifies that the packet should be dropped and not processed further. NF_ACCPET signifies that the packet should be accepted and passed to higher layers. NF_STOLEN signifies that the packet should be accepted but not passed to higher layers. NF_QUEUE puts the packet into the Netfilter queue. NF_REPEAT repeats the hooked function again.

The hooks given in Table 2 populate the socket kernel buffer, sk_buff structure. The sk_buff structure is the data type by which packets are represented within the kernel. Each packet has its own sk_buff structure.

The sk_buff structure is shown below in shorthand for clarity and found in *<linux/skbuff.h>*.

```
struct sk_buff {
.
.
/* Transport layer header */
Union
        {
                struct tcphdr   *th;
                struct udphdr   *uh;
                struct icmphdr  *icmph;
                struct igmphdr  *igmph;
                struct iphdr    *ipiph;
                struct spxhdr   *spxh;
                unsigned char
        *raw;
        } h;

/* Network layer header */
Union           {
                struct iphdr    *iph;
                struct ipv6hdr  *ipv6h;
                struct arphdr   *arph;
                struct ipxhdr   *ipxh;
                unsigned char   *raw;
        } nh;
.
.
};
```

Figure 5 Important fields in the sk_buff structure.

According to [6] sk_buff contains pointers to headers of the TCP/IP stack. The unions h, nh, mac represent layer 4, layer 3 and layer 2 respectively. Each of h, nh, mac contains structures for each protocol.

When a packet is received the layer *n* function receives skb->data from the layer *n-1* below. The respective pointer is then initialised for layer *n i.e.,* skb->nh for Layer 3.

Once processing has been completed the skb->data is updated to point to the beginning of the *n+1* layer. Again according to [6] sending a packet reverses the above procedure.

The two important structures within the transport and network layer structures are shown in Figure 6 below and are found within the kernel source code in the files <*linux/tcp.h*> and <*linux/ip.h*>.

```
struct tcphdr  *th;
struct iphdr   *iph;
```

Figure 6 Linux kernel network stack transport and network layer structures.

Within the structure iphdr we have the following interesting members:

```
struct iphdr {
.
.
        __u32   saddr;
        __u32   daddr;
.
.
};
```

Figure 7 Linux kernel network stack network layer structure showing key members.

A very important aspect of Intrusion Detection is knowing where the attacks seem to be originating from. Here, saddr and daddr are the packet's source and destination IP addresses respectively.

These can be called within our network data collector as nh.iph->saddr and nh.iph->daddr respectively. Our network data collector at this moment has at its disposal the originating IP of the packet.

The next important aspect of Intrusion Detection is knowing which port and hence service the packet is destined to. Looking at struct tcphdr in *<linux/tcp.h>*

```
struct tcphdr {
.
.
        __u16   dest;
.
.
};
```

Figure 8 Linux kernel network stack transport layer structure showing key member.

We see that the port the packet is destined to can be easily looked up by the network data collector through h.th->dest. We are now in a position whereby we know the source IP and the port the packet is destined to, the next step is determining what the packets payload is. This is given below from struct tcphdr casting as type char:

```
data = (char *)((int)tcp + (int)(tcp->doff * 4));
```

Figure 9 TCP/IP packet application layer data.

We now have access to the packets payload and are in a position whereby we can disassemble each and every incoming packet and inspect its payload to see if it contains malicious content.

Being a signature based NIPS, pattern matching within the payload is the most computationally intense task as well as the task requiring the largest memory requirements. This because every packet's payload must be compared against every single rule base to see if a match exists.

In our case the Network Data Collector will hook the NF_IP_POST_ROUTING hook. This shall be accomplished through the use of a Linux loadable kernel module *i.e.*, an LKM. The reader can find more information about LKMs and their programming in [30].

In the following chapters we will create our own mock attack from the ground-up using the same concepts that are used by attackers in creating exploits. We will then create an indicative signature for this attack which we will program our NIPS to detect and protect from.

6.1.1.1 Vulnerabilities and Exploits

According to [23] a vulnerability is a software hardware or procedural weakness that may allow a adversary the chance they are looking for to gain unauthorized access on a computer system. Vulnerabilities may be found in *e.g.*, an old unpatched daemon running on a machine, within the operating system itself or unpatched applications.

A threat is defined as someone, the threat agent, who will discover the existence of this vulnerability and try to exploit it. Exploitation will either be done using custom crafted tools or those created by others and will either be manual requiring a larger skill-set from the side of the attacker or automated using tools such as Core Impact[g] or Metasploit[h].

A risk is defined again according to [23] as the likelihood of the threat agent successfully exploiting the vulnerability and the associated business impact this may have.

At any given time a computer system may be vulnerable to a number of vulnerabilities which may be successfully exploited by a threat agent. Therefore in order to reduce risk to acceptable levels *i.e.*, the businesses risk appetite[i] , within which the business or end-user is comfortable operating within, countermeasures are installed.

Countermeasures aim to actively or passively mitigate vulnerabilities and hence make identifying vulnerabilities, exploiting them and even leveraging access to higher privilege levels harder for the attacker.

---

[g] Core Impact is a commercial automated penetration testing suite aimed at information security professionals and can be found at http://www.coresecurity.com/

[h] Likewise Metasploit is also a commercial-grade yet free penetration testing suite which can be found at http://www.metasploit.com

[i] Risk Appetite is the risk exposure in monetary terms that a business is comfortable operating within and which would not be of a significant burden to business continuity in the event of a disaster.

The network and host IPS are such countermeasures which will be installed within a corporate network to mitigate the vulnerabilities that the information systems sitting behind it and those on which it is running are exposed to.

The IPS being signature based must identify and thwart exploitation attempts within the incoming TCP/IP traffic. To accomplish this, we now describe buffer overflow vulnerabilities and having understood these, we will create our own signature for an example mock attack we shall create.

Having understood how to create a signature for this network attack using the same concepts it is easy to extrapolate and create similar signatures for other vulnerabilities.

6.1.1.2 Buffer overflow vulnerabilities

Stack based buffer overflows according to [28] are the most prevalent type of vulnerabilities which occur in the C programming language. Despite widespread knowledge about their occurrence and how to avoid buffer overflow conditions through secure programming, they still resurface today.

In C programming, a buffer is defined as a continuous array of allocated memory of a specific size. This is allocated within a C program as in Figure 10 below.

```
char buffer[10];
```

Figure 10 Declaring a character buffer in the C programming language.

Figure 10 tells the compiler to reserve 24 bytes on the stack. We can verify this by compiling a simple program using the declaration in Figure 10 and then disassembling the program with the GNU debugger *gdb* and seeing how many bytes are reserved on the stack. A simple program using the declaration in Figure 10 is given in Figure 11 below.

```
#include <stdio.h>

int main (int argc, char **argv) {
      char buffer[10];
      strcpy(buffer,argv[1]);
      printf ("You said: %s\n",buffer);
}
```

Figure 11 Simple program declaring a buffer of size 10 bytes, which in the stack is allocated as 24 bytes.

We now go on to compile the program given in Figure 11 and then disassemble it using *gdb*. This is shown in Figure 12.

```
bash-3.00#  gdb ./exit
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i486-slackware-linux"...Using host
libthread_db library "/lib/libthread_db.so.1".

(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   %ebp
0x080483c5 <main+1>:    mov    %esp,%ebp
0x080483c7 <main+3>:    sub    $0x18,%esp
.
.
.
```

Figure 12 Disassembly of C language program given in Figure 11 showing that 24 bytes have been allocated on the stack to accommodate char buffer [10].

The sub $0x18,%esp instruction according to [28] reserves 24 bytes on the stack *i.e.*, 0x18 hexadecimal in binary is twenty four bytes.

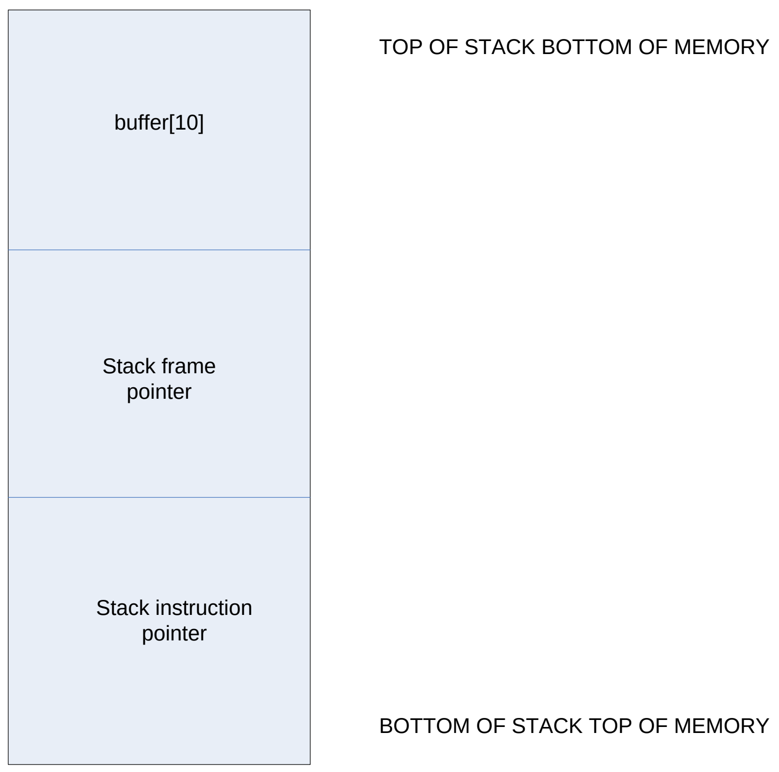

buffer[10]
Stack frame pointer
Stack instruction pointer
TOP OF STACK BOTTOM OF MEMORY
BOTTOM OF STACK TOP OF MEMORY

Figure 13 Layout of stack right after buffer[10] has been allocated.

The stack at this moment in time is laid out as in Figure 13. The C programming language makes no effort to do bounds checking rather leaves this to the programmer *i.e.*, checking that the amount of data copied into buffer[10] does not exceed the amount of memory allocated; in this case twenty four bytes.

Therefore, should the programmer not be careful in their string manipulation and copy in to buffer[10] more than twenty four bytes a condition known as a buffer overflow occurs.

Buffer overflows have been long known and exploited in the underground community *i.e.*, the Morris Worm in 1988 [29], however the first ever formal description of their occurrence, exploitation and prevention came much later in 1996 in the article in the underground Phrack magazine called Smashing the Stack for Fun and Profit by Aleph One [38].

Functions such as the standard C library function *strcpy* can be used to fill char buffer[10] past its twenty four bytes hence overflowing the allocated twenty four bytes and overwriting adjacent memory locations; the stack frame pointer and the stack instruction pointer. This would be done by the following code:

```
strcpy(buffer, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA);
```

Figure 14 Use of the strcpy function to overflow char buffer[10] ‘s allocated memory on the stack, 24 bytes, and hence overwrite adjacent memory locations.

What is happening in Figure 14 is that we are filling the buffer with thirty two bytes; twenty four to reach the twenty fourth byte allocated by char buffer [10], four bytes to overwrite the stack frame pointer and a further four bytes to overwrite the stack instruction pointer.

We have therefore overwritten both stack frame pointer and stack instruction pointer with arbitrary data. The stack now looks like Figure 15.

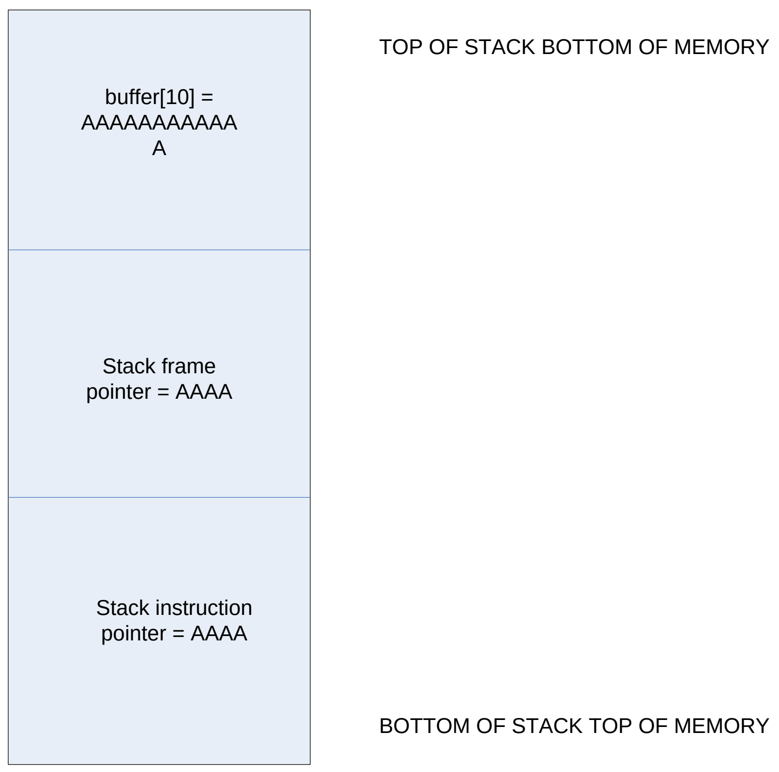

TOP OF STACK BOTTOM OF MEMORY
buffer[10] = AAAAAAAAAAA A
Stack frame pointer = AAAA
Stack instruction pointer = AAAA
BOTTOM OF STACK TOP OF MEMORY

Figure 15 Layout of stack right after buffer[10] and adjacent registers the stack frame pointer and stack instruction pointer have been overwritten.

We can also verify that we have overwritten both the stack frame pointer and stack instruction pointer by running our program given in Figure 11 with 32 character input. This is shown in Figure 16 below.

```
bash-3.00#  ./x AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You said: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
bash-3.00#
```

Figure 16 Executing program in Figure 11 with thirty two byte input which gets copied in to a 24 byte allocated buffer. Therefore, overflowing the stack frame pointer and stack instruction pointer and hence segfaulting.

What we see from Figure 16 above is that when our program is run with a 32 character input, the *strcpy* function copies the input into the twenty four byte buffer, hence overflowing and overwriting the adjacent registers; the stack frame pointer and stack instruction pointer.

According to [26] the instruction pointer contains the offset in the current code segment for the next instruction to be executed. On Intel IA32 processors the instruction pointer is called EIP *i.e.*, extended instruction pointer.

The program exits abruptly with a segmentation fault (segfault) because the instruction pointer according to [38] and [28] determines the flow of execution of the program. Hence by overwriting the EIP with an invalid memory address, the next instruction to be executed is invalid, hence a segfault occurs.

We can use the GNU debugger *gdb* to see what the current values of the stack frame pointer and stack instruction pointer are. This is shown in Figure 17 below. What we notice is that both the stack frame pointer, EBP and stack instruction pointer EIP contain the values 0x41414141. 0x41 is the hexadecimal ASCII equivalent of the character A *i.e.*, the character we used in Figure 16 to overflow the stack.

```
bash-3.00#  gdb -c core ./x
Core was generated by `./x AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb) i r
eax            0x2b     43
ecx            0x40148840       1075087424
edx            0x2b     43
ebx            0x40147ff4       1075085300
esp            0xbffff730       0xbffff730
ebp            0x41414141       0x41414141
esi            0xbffff790       -1073743984
edi            0x2      2
eip            0x41414141       0x41414141
```

Figure 17 *gdb* program output showing the current state of the stack frame pointer and stack instruction pointer, EBP and EIP respectively. The program has terminated with a Segmentation fault because EIP has been overwritten with an arbitrary memory location 0x41414141 where execution tried to next jump to.

Therefore due to the no bounds checking done by the *strcpy* function we are in a position whereby we are able to overwrite the allocated buffer and control what values get written to the instruction pointer and hence control the flow of execution of the program; redirecting execution to any area; preferably those controlled by the attacker for reasons made clear below, within the process memory space.

The end result of a buffer overflow is the modification of the flow of execution of a program. This is achieved by the modification of the instruction pointer address which is made to point somewhere within the process memory space where the attacker has stored their shellcode. This point within the process memory space in our example was the 0x41414141 value of the instruction pointer, EIP.

To reiterate, buffer overflows are considered one of the most frequent type of attacks. What we have done thus far therefore is to explain how they occur and how they can be abused by an attacker to modify the flow of execution of a program.

The next section will describe shellcode which constitutes the set of executable instructions which execution flow is redirected to. It also forms the section of the malicious payload which is sent with each exploit. Hence the segment of a malicious TCP/IP packet which our NIPS will try to detect when we describe signature generation in Chapter 6.1.1.4.

6.1.1.3 Introduction to shellcode

Shellcode according to [38] and [28] is a set of machine opcode instructions that are injected and then executed by a exploited program. Traditionally shellcode was used to give the attacker a console shell on the exploited machine, hence its name.

Shellcode is the portion of the malicious payload which our NIPS will try to detect amongst all the other legitimate traffic, therefore understanding what it is and how it works is essential to creating robust signatures to detect attacks.

We will focus on Intel x86 IA32 processors [26] and Linux. The concepts that follow are the same and transferable should we be discussing other processor architectures such as Irix MIPS and Sun SPARC and other operating systems such as Microsoft Windows and other UNIX flavors.

What shellcode does according to [28] is execute a given system call with a set of arguments. The integer number of the system call required is stored in the EAX register. A list giving all integers and corresponding system calls can be found in *<sys/syscall.h>*.

The arguments of the system call are then stored in EBX, ECX, EDX, ESI, EDI registers. Finally, the system call is executed by calling the int 0x80 instruction.

To put things in to perspective, we will create a simple shellcode to demonstrate how these concepts fit together. In Figure 18 below, we use C in-line assembly programming to invoke the *sys_exit* system call. Our ulterior aim is to use this program to create a shellcode which will execute *sys_exit*.

```
#include <stdio.h>
int main (int argc, char **argv) {
__asm__(
       "xor %eax,%eax\n"
       "xor %ebx,%ebx\n"
       "inc %eax\n"
       "int $0x80\n"
       );
}
```

Figure 18 Using C programming language in-line assembly to call the *sys_exit* system call.

In Figure 18 we nullify the EAX and the EBX registers by xoring them against each other. XORing two quantities which are alike always gives a result of zero. We then increment the EAX register. By so doing, the EAX register now equals one and the EBX register contains the value zero.

We now remind ourselves that the integer number of the system call we require is stored in the EAX register. The integer number one corresponds to the *sys_exit* system call as defined in <*sys/syscall.h*> and given as a prototype below:

```
asmlinkage long sys_exit(int error_code);
```

Figure 19 The *sys_exit* system call prototype.

The variable *int error_code* is an integer quantity that denotes if the program exited smoothly or whether it exited abnormally. An *error_code* of zero denotes that the program exited smoothly.

We also remember that arguments of system calls are stored sequentially in EBX,ECX, EDX, ESI, EDI respectively. Hence the *error_code* shall be stored in the EBX register. Since we would like our shellcode to pretend it has exited smoothly simply setting EBX to zero (hence the XOR of %ebx,%ebx), does this for us. Finally, we call interrupt 0x80 hex which executes the system call. The output of the program once executed is shown below:

```
localhost:/tmp# ./exit
localhost:/tmp#
```

Figure 20 Execution of the assembly instructions given in Figure 18.

Disassembling our program *exit* using the GNU debugger *gdb* we get the output given in Figure 21.

```
localhost:/tmp# gdb ./exit

      (gdb) disassemble main
      Dump of assembler code for function main:
      .
      .
      0x08048332 <main+14>:   xor    %eax,%eax
      0x08048334 <main+16>:   inc    %eax
      0x08048335 <main+17>:   xor    %ebx,%ebx
      0x08048337 <main+19>:   int    $0x80
      .
      .
      End of assembler dump.
      (gdb) quit
localhost:/tmp#
```

Figure 21 GNU debugger output, disassembly of our program *exit* which demonstrates the use of in-line assembly to create shellcodes.

We can see from Figure 21 that at main+14 to main+19 we have the same assembly code as we did in our program in Figure 18. To create shellcode from the above disassembly we shall use the GNU program called *objdump*. The output is given in Figure 22.

```
localhost:/tmp# objdump --disassemble-all ./exit

      08048324 <main>:
      .
      .
       8048332:       31 c0            xor    %eax,%eax
       8048334:       40               inc    %eax
       8048335:       31 db            xor    %ebx,%ebx
       8048337:       cd 80            int    $0x80
      .
      .
localhost:/tmp#
```

Figure 22 Objdump disassembly of our *sys_exit* program showing opcodes which correspond to xor, inc and int instructions.

The second column in Figure 22 is the opcode for each assembly instruction. Therefore we can infer that 31 c0 is the opcode for the xor %eax,%eax instruction. We also see two lines down that 31 db is the opcode for the xor %ebx,%ebx instruction. Therefore we infer that opcode 31 signifies XOR, and this is the case for x86 IA32 Intel processors.

To transform Figure 22 into a real working shellcode suitable for use in an exploit written in the C programming language, we must represent the opcode instructions as follows:

```
char shellcode[] = “\xc1\xc0\x40\x31\xdb\xcd\x80”;
```

Figure 23 Linux *sys_exit* Intel IA32 Shellcode

This string must be free from nulls *i.e.,* \x00. This because when we try to overflow the buffer in the stack overflow we are copying the shellcode into a character buffer. \x00 signifies the end of a string and hence all our shellcode would not be copied into our buffer.

Getting rid of \x00 characters is a matter of more elegant programming and replacing *e.g.,* mov $0x0, %ebx with a call such as xor %ebx, %ebx or using the low and high 8-bit sections of the 32 bit extended registers *e.g.,* instead of mov $0x0,%ebx we can use movb $0x0,%bl. This saves 0x0 into the lower 8-bit section of the 32 bit register EBX[26],[28].

We have demonstrated how to create a working shellcode for the *sys_exit* call given in Figure 23. So, when the stack is overflowed, the return address is overwritten and points to the address within the process memory space where our shellcode is stored. This is usually within the buffer itself. Once the shellcode is executed, *sys_exit* would be called with *int error_no* = 0 *i.e.,* the exploited process would exit normally.

This is a simple example of a shellcode simply to demonstrate the rationale behind their creation and use. Attackers usually will create shellcodes with combinations of system calls which would for example, drop privileges of the process to the lowest possible level *i.e.,* root.

Dropping privileges of the process to the lowest possible level, root, is accomplished by setting the *sys_setuid* and *sys_setgid* system calls both to zero *i.e., sys_setuid(0)* and *sys_setgid(0).*

Having achieved this, through the shellcode, the attacker would then use the *sys_execve* system call to execute a console such as */bin/bash* – the Linux bash shell. This would execute the bash shell with root privileges, hence full root console access on the target machine.

Having secured access on the machine, the attacker can proceed to attack other machines within the corporate network.

The next section shall describe how attack signatures are created for network intrusion protection systems, and we will create an indicative signature for our *sys_exit* shellcode.

6.1.1.4 Signature generation for Network Intrusion Protection

According to [10] signature based IDS systems are the most common types of IDS in both commercial and research implementations. They however require prior knowledge of attack patterns *i.e.*, signatures. This because signature based IDS's match observed traffic against attack patterns to differentiate between legitimate and malicious traffic.

In the case of network intrusion protection, signatures can constitute two types [10]; those detecting rate-based attacks[j] *e.g.,* number of SYN connections within a given timeframe which if exceeded would mean that the host is under SYN flood attack and those signatures which portray malicious payloads within the packets themselves. Our implementation will focus on the latter case.

The key to creating robust signatures lies within detecting portions of malicious payloads that remain constant no matter the different strains of the attack.

Unfortunately according to [9] and [43] even widely deployed IDS systems such as Sourcefire's Snort utilise signatures within their detection routines which are not generic enough to catch variants of the same attack or use signatures which rely on circumstantial conditions.

[j] Rate-based signature detection is also called packet distribution detection.

This ultimately leads to the generation of many false positives *i.e.*, detected attacks that are actually false alarms.

Creating a good attack signature is a fine balance between specificity *i.e.*, how tuned the signature is to a specific attack and generality *i.e.*, how many other similar attacks that same signature is able to detect. Furthermore according to [43], it is also a balance between the number of failed attacks and those which actually have an impact.

False positives can significantly burden the management effort required by the IDS analyst to sort through failed probing attempts, distracting from events which are more critical and hence require immediate attention.

For an indication of the extent of network probing in [16] the Code Red worm is described to have infected approximately 2000 hosts per minute at its peak. Although this is now a phased out vulnerability, to this date, one still witnesses an alarming number of Code Red infection attempts within web server log files.

We will now create a signature to detect our example shellcode given in Figure 23. The signature which we will create shall be generic enough to catch simple variations of the *sys_exit* shellcode as well as other attacks incorporating the *sys_exit* shellcode within one of their attack payload stages.

Referring to Figure 22, the *objdump* output for our *exit* program, the section of the shellcode which would always be the same and not vary is the xor %eax,%eax instruction followed by the inc %eax instruction. This because for *sys_exit* to be called by interrupt 80, EAX must always contain the value of one.

```
localhost:/tmp# objdump --disassemble-all ./exit

      08048324 <main>:
      .
      .
       8048332:       31 c0            xor    %eax,%eax
       8048334:       40               inc    %eax
       8048335:       31 db            xor    %ebx,%ebx
       8048337:       cd 80            int    $0x80
      .
      .
localhost:/tmp#
```

Figure 24 Objdump disassembly of our *sys_exit* program showing opcodes which correspond to xor, inc and int instructions.

Looking up the respective opcodes for these assembly instructions from Figure 22, we see that \x31\xc0 is the opcode for xor %eax,%eax, \x40 is the opcode for inc %eax and \xcd\x80 is the opcode for int $0x80. Therefore for us to detect the *sys_exit* shellcode we have the following pseudocode for a rule.

```
if (\x31\xc0\x40 is in traffic) AND (\xcd\x80 is also in traffic)
{
                  sys_exit shellcode detected
}
```

Figure 25 Signature to detect our *sys_exit* shellcode within a network-based attack.

The opcode cd 80 represents int 80. Many intrusion detection systems look out for the cd 80 string as it always means[k] that an exploitation attempt has been made and shellcode sent to the daemon [13]. It therefore frequently forms a generic catch-all rule by itself for various strains of buffer overflow attacks.

Now that we have created a signature to detect our shellcode, we shall incorporate this within our NIPS. We shall now create the routine which protects the system once the shellcode is detected by our signature in Figure 25. This is described in the next section.

6.1.1.5 Invoking iptables from kernel space

According to [51] methods of pro-active NIPS countermeasures include updating firewall rules and dropping connections. Our NIPS implementation shall use the former case as this is deemed to pose a more permanent solution to mitigate attack attempts.

However, one of the main disadvantages of updating firewall rules to block attackers according to [51] is that an attacker can spoof their source IP with relative ease hence making such approaches not as fool proof and prone to being used as a DoS against company servers[l].

---

[k] On x86 Intel Processors.

[l] An attacker can spoof the source IP of his attack to make the origin of the attack seem like a critical company server. This would cause the IPS to effectively block all traffic to and fro that machine, hence effectively acting as a DoS attack.

However, the threat of an attacker spoofing the source IP of an attack is effectively minimised as we only accept full-duplex connections. This is the same approach described in [51].

The NIPS system is exclusively based on the Netfilter framework which is itself the kernel component of the popular *iptables* Linux firewall. *Iptables* comes installed with all modern Linux distributions which run the 2.6 series kernel.

Much like Netfilter, *iptables* possesses INPUT rules, FORWARD rules and OUTPUT rules as shown in Figure 4. These dictate what security policies are effective for packets arriving, transversing and leaving a given host.

Once we detect the *sys_exit* shellcode given in Figure 23, we shall invoke *iptables* from kernel space so as to update its rule set to deny further connections from the attacker.

The two rules we shall be using are INPUT and OUTPUT rules. Many shellcodes conventionally invoke a bindshell. This effectively means that a TCP port is opened on the machine which listens for incoming connections. Once one is made, then the attacker is presented with a fully featured console.

By adding the attackers source IP within *iptable's* INPUT filtering rule, we effectively block all inbound connections originating from him. Therefore, even if his shellcode has been successful in exploiting the daemon and binding a console on a TCP port, his connection attempt to that port will be thwarted.

A sophisticated attacker however could utilise a shellcode which sends the console back to a port listening on his own machine. Therefore effectively bypassing our INPUT filtering policy as an outbound connection is initiated.

To counteract this threat as well, we also add the attackers IP within the OUTBOUND filtering policy. We therefore thwart all outbound connection attempts to the attackers IP.

To invoke *iptables* within kernel space, we use the *call_usermodehelper* function. *Iptables* is spawned as a child of a kernel thread called *keventd* [30]. The function which does this is given in Figure 26.

```
void worker(void *stuff) {

      char *argv[8];
      char *envp[3];
      char src[17];
      strcpy(src, __ntoa((unsigned long)global_ip));
      argv[0] = "/sbin/iptables";
      argv[1] = "-A";
      argv[2] = "INPUT";
      argv[3] = "-s";
      argv[4] = src;
      argv[5] = "-j";
      argv[6] = "DROP";
      argv[7] = (char *)0;
      envp[0] = "HOME=/";
      envp[1] = "PATH=/:/sbin:/bin:/usr/sbin:/usr/bin";
      envp[2] = (char *)0;
      call_usermodehelper(argv[0],(char **)argv,envp,1);
      argv[0] = "/sbin/iptables";
      argv[1] = "-A";
      argv[2] = "OUTPUT";
      argv[3] = "-d";
      argv[4] = src;
      argv[5] = "-j";
      argv[6] = "DROP";
      argv[7] = (char *)0;
      envp[0] = "HOME=/";
      envp[1] = "PATH=/:/sbin:/bin:/usr/sbin:/usr/bin";
      envp[2] = (char *)0;
      call_usermodehelper(argv[0],(char **)argv,envp,1);
}
```

Figure 26 The NIPS protection function which invokes *iptables* from within the kernel using *call_usermodehelper*.

We see from Figure 26 that *call_usermodehelper* is invoked twice, once to add the attackers IP within the *iptables* INPUT filtering policy, and once to add the attackers IP within the *iptables* OUTPUT filtering policy.

To add the attackers IP within the INPUT filtering policy we invoke *iptables* with the following options:

```
iptables –A INPUT –s IP –j DROP
```

Figure 27 Use of iptables to deny all inbound connections from an attackers IP.

This tells *iptables* to deny all inbound connection attempts using all protocols *e.g.,* ICMP, UDP, TCP.

To add the attackers IP within the OUTPUT filtering policy we invoke *iptables* with the following options:

```
iptables –A OUTPUT –d IP –j DROP
```

Figure 28 Use of iptables to deny all outbound connections to an attackers IP

This tells *iptables* to drop all outbound connection attempts where the destination IP is the IP of the attacker indiscriminate of the type of protocol used *i.e.,* ICMP, UDP, TCP.

We now compile and install the NIPS and demonstrate how it achieves all the above in practice.

Compiling the NIPS is a simple matter of running the GNU *make* command. *Make* finds and links all dependencies, and creates the loadable kernel module. Once the loadable kernel module has been created *i.e., nips.ko,* we load it within the kernel. This is done using the insert module, *insmod,* command.

```
bash-3.00# make
make -C /lib/modules/2.6.15/build M=/root/nips2 modules
make[1]: Entering directory `/usr/src/kernels/2.6.15'
  CC [M]  /root/nips/nips.o
  Building modules, stage 2.
  MODPOST
  CC      /root/nips/nips.mod.o
  LD [M]  /root/nips/nips.ko
bash-3.00#
bash-3.00# insmod nips.ko
bash-3.00#
```

Figure 29 Installation of NIPS

We will now demonstrate the attack. We will connect to the Apache web server running on port 80. Here, we will mount a hypothetical attack whereby our injected shellcode is able to be passed through the URL parameter of the website residing on the server. Our shellcode string in printable ASCII is represented by the `1□@`` characters. This is sent to the daemon as shown in Figure 30.

While we are performing this attack, the NIPS is carefully examining all TCP/IP traffic and inspecting all application-layer payloads for possible occurrences of our shellcode.

```
bash-3.00# telnet 192.168.0.2 80

Trying 192.168.0.2...
Connected to 192.168.0.2.
Escape character is '^]'.
GET /somedirectory/1□@` HTTP/1.0

HTTP/1.1 404 Not Found
Date: Wed, 31 Jan 2007 12:56:19 GMT
Server: Apache/2.0.54 (Unix)
.
.
bash-3.00#
```

Figure 30 Attacking the Apache web server with our sys_exit shellcode

The response elicited by the NIPS is instantaneous. *Iptables* is invoked adding the attackers IP to both INPUT and OUTPUT chains and a management message is placed in the administrative log file */var/log/messages* warning the root user that an attack has occurred. This is shown in Figure 31.

```
bash-3.00# tail /var/log/messages
Jan 31 12:55:49 localhost kernel: Accepted http request
Jan 31 12:56:19 localhost kernel: Detected sys_exit shellcode!!
bash-3.00# iptables --list
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  192.168.0.1          anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  anywhere             192.168.0.1
bash-3.00#
```

Figure 31 The intrusion protection response, the attacker, 192.168.0.1 has been firewalled automatically by *iptables* in both INPUT and OUTPUT chains. Hence effectively mitigating further progression of the attack.

As can be seen in Figure 31 we have successfully mitigated the network centric attack. This therefore brings the NIPS chapter to a close. We will now describe the host intrusion protection system in the next chapter.

6.1.2 Host Intrusion Protection System

As described in chapter 5.0, the HIPS will hook the *execve* library call so that every time a program is executed from the console our own *execve* will handle running the program.

Our own *execve* will create an MD5 hash of the binary on runtime and compare this generated hash with that stored. If the two hashes match, then execution will be allowed to commence otherwise, access to the binary will be denied. The flowchart explaining this is shown in Figure 32 below.

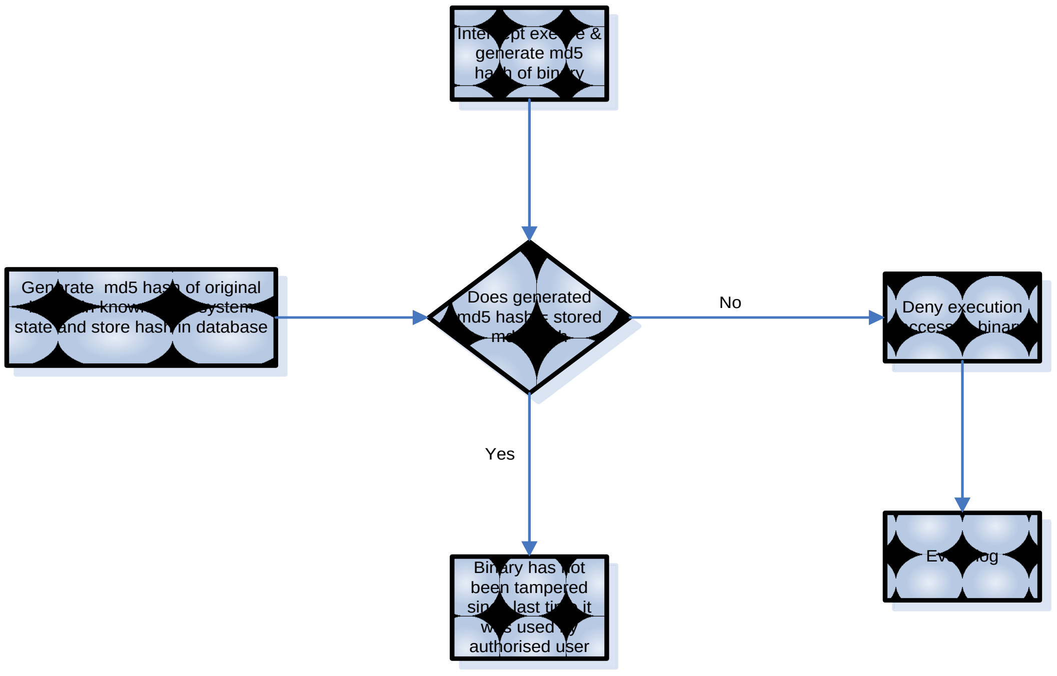

Intercept execve & generate md5 hash of binary
Generate md5 hash of original known system state and store hash in database
Does generated md5 hash = stored md5 hash
No
Deny execution access to binary
Yes
Event log
Binary has not been tampered since last time it was used by authorised user

Figure 32 A flowchart depicting the Host Intrusion Protection System. The *execve* system call is hooked, replacing it with our own which creates MD5 hashes of the binary on run time. These MD5 hashes are matched against those stored in a database.

According to [27] an executable file is either dynamically linked or statically linked. Statically linked executables are self-contained and thus can execute on a machine other than the one on which the binary was compiled on, thus offering portability.

Dynamically linked executables on the other hand are dependant on shared libraries to function. Statically linked executables are larger in size, dynamically linked executables are smaller in size but unless they are executed on a system which possesses the same libraries and their versions as the system on which the binary was compiled on, they can not run.

Most executables on a given Linux system are dynamically linked. The GNU C language compiler *gcc* compiles executables dynamically by default, statically linked executables must be compiled explicitly with *gcc* with the *–static* option.

Figure 33 below shows the *ldd* output for the */bin/ls* directory listing program. *ldd* is used to print out the shared libraries that a given dynamically linked executable is compiled against.

```
localhost:/tmp# ldd /bin/ls
linux-gate.so.1 => (0xffffe000)
librt.so.1 => /lib/tls/librt.so.1 (0xb7fd7000)
libc.so.6 => /lib/tls/libc.so.6 (0xb7eba000)
libpthread.so.0=> /lib/tls/libpthread.so.0 (0xb7ea8000)
/lib/ld-linux.so.2 (0xb7feb000)
localhost:/tmp#
```

Figure 33 *ldd* program output showing the shared libraries which are related to the */bin/ls* directory listing program.

We will be using library interposition which is described in [5] and in [61] to implement our HIPS. This allows the HIPS to perform a man-in-the-middle attack between a dynamically linked executable and the shared objects it references. Schematically this is described in Figure 34 below.

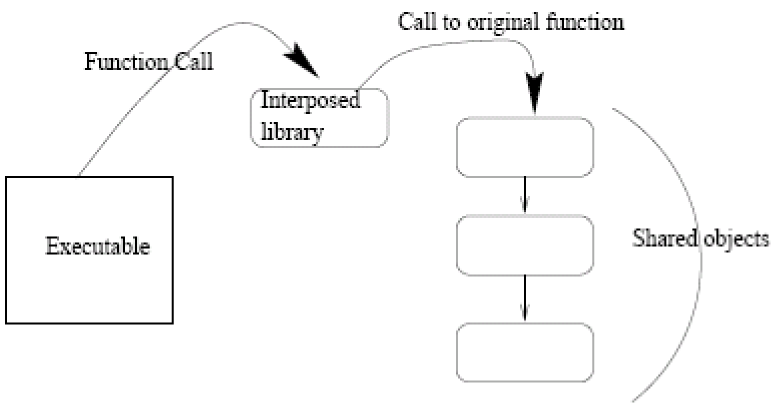


Figure 34 From [5] describing the method of library interposition. We see above that an dynamically linked executable calls a function call which is intercepted and processed by the interposed library instead of that referenced hence performing a kind of a man-in-the-middle attack.

In our case, we will be intercepting the *execve* function which exists within the standard C library, *libc.so.6*. In relation to Figure 34, *execve* would lie within the right-hand side of the diagram within the shared objects (*libc.so.6*); it is an original function.

According to [5] the runtime linker joins a dynamically compiled program with its associated libraries on run-time *e.g.,* those referenced in Figure 33 above in the case of */bin/ls*. Again according to [5] both Linux and Solaris possess an environment variable called LD_PRELOAD which allows a shared object to be interposed.

When an external function is called within the executable and the LD_PRELOAD environment variable is set, the runtime linker will first try to find occurrences of the function within the library specified by the LD_PRELOAD environment variable. If the called function exists within the interposed library then it is executed instead.

Using library interposition instead of system call interposition according to is preferred because library interposition can intercept standard C library functions.

In [61] using the example of the system call read, *sys_read* and the standard C library functions *gets*, *scanf* which interface to it, if we wanted to intercept only scanf using system call interposition we would intercept *sys_read* which would mean that all functions that rely on *sys_read e.g.,* gets would be affected as well. Therefore, less overhead is placed on the system by using library interposition which allows us to selectively hook higher level functions.

[61] Goes on to say that the main weakness of using library interposition is that a skilled attacker can disable the interposed library hence in our case rendering our integrity checks void.

Whilst this is true, the same goes for system call interposition when done through hooking system calls through Linux kernel modules (LKM). Disabling the library interposition is a matter of either uploading a statically compiled binary equivalent of the binary we wish to overwrite, unseting the LD_PRELOAD environment variable or blanking out the */etc/ld.so.preload* entry. Equivalently for system call interposition, simply typing *rmmod* followed by the name of the LKM would achieve the same effect.

It is therefore inevitable in the face of a skilled attacker that our system can be bypassed. However as we are intercepting the compromise attempt prior to itself being realised and protecting from it as in [61] the issue of bypassing the library interposition is a non-issue.

So, what we have achieved until now is that we have a method by which we can interpose the standard C library *libc.so.6* and intercept calls to the *execve* function by just setting the environment variable LD_PRELOAD=./our-host-intrusion-protection-execve.so globally.

We now must create our own *execve* function which will intercept all execution attempts. The *execve* function's prototype is defined in the Linux manual page for *execve* and is shown in Figure 35 below.

```
int execve(const char *filename, char *const argv [], char
*const envp[]);
```

Figure 35 Showing the *execve* function prototype. The first argument is the file name, the second the executed programs arguments and the third, and the current environment variables, of which LD_PRELOAD is one.

Our own *execve* function in pseudocode will be performing the following operations:

1. Retrieve the name of the filename const char *filename, arguments char *const argv[] and environment variables char *const envp[].
2. We will then open the standard C library and retrieve a pointer to the real execve and store this as realexecve.
3. We will then open a file pointer to the filename const char *filename and use this to create an MD5 sum of const char *filename.
4. We then compare the stored MD5 sum with that of the MD5 sum we have pre-computed for that specific binary.
5. If MD5 sums match, then access allowed, otherwise violation is logged and denied.
6. Denying access basically means we swap filename with */bin/denied i.e.*, execution redirection.

In Figure 36, the HIPS was compiled, we then went on to set the LD_PRELOAD environment variable. This automatically as previously discussed redirected all *execve* requests to our own *execve* function.

Within our function, we had computed an MD5 hash for the UNIX secure shell program. When *ssh* is run, we get the output we would expect, this because the program has not been modified or altered since the MD5 hash was created; it's secure to use.

We then attempt to maliciously overwrite the *ssh* program with the */usr/bin/id* program which tells us our current user level. We see that despite ourselves having root access *i.e.*, superuser, when we try to run *ssh* again, our access to it is denied therefore protecting us from possibly running malicious/trojan code.

```
bash-3.00# ls
hips.c
bash-3.00# gcc -fPIC -shared hips.c -o hips.so
bash-3.00# export LD_PRELOAD=/hips.so
bash-3.00# ssh
usage: ssh [-1246AaCfgkMNnqsTtVvXxY] [-b bind_address]
            [-c cipher_spec]
           [-D port] [-e escape_char] [-F configfile]
           [-i identity_file]
           [-L [bind_address:]port:host:hostport]
           [-l login_name] [-m mac_spec] [-O ctl_cmd]
           [-o option] [-p port]
           [-R [bind_address:]port:host:hostport] [-S ctl_path]
           [user@]hostname [command]
bash-3.00# id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
bash-3.00# cp /usr/bin/id /usr/bin/ssh
bash-3.00# ssh
Access to that file has been denied please contact your security
administrator. This event has been logged.
bash-3.00#
```

Figure 36 Showing the HIPS in action. Access to the overwritten Unix secure shell program (*ssh*) is denied despite ourselves possessing root level access.

As can be seen in Figure 36 we have successfully mitigated the attack. We will now introduce the concept of Intrusion Remediation Systems.

6.1.2.1 Host Intrusion Remediation System

In [15] the window of time between an attack being detected and a protective response being executed is described. If a skilled attacker is given ten hours subsequently to being detected they will be successful in furthering their attack 80% of the time.

This increases to 95% success should the attacker be given twenty hours and finally success of further corporate LAN penetration is certain should the attacker be given thirty hours.

Therefore to reduce the window of opportunity to the absolute minimum, and hence proportionally reducing the success rate of the skilled attacker, we propose an intrusion remediation system.

In such a case, taking corrective action would consist of first detecting the impeding attack, then protecting from it. Up until this stage we have done the exact same actions as in our simple HIPS. The remediation stage would involve copying back a known good copy (from *e.g.,* immutable media) of the binary. This realised concept is shown in Figure 37 below:

```
bash-3.00# ls
hips.c
bash-3.00# gcc -fPIC -shared hips.c -o hips.so
bash-3.00# export LD_PRELOAD=/hips.so
bash-3.00# ssh
usage: ssh [-1246AaCfgkMNnqsTtVvXxY] [-b bind_address]
            [-c cipher_spec]
           [-D port] [-e escape_char] [-F configfile]
           [-i identity_file]
           [-L [bind_address:]port:host:hostport]
           [-l login_name] [-m mac_spec] [-O ctl_cmd]
           [-o option] [-p port]
           [-R [bind_address:]port:host:hostport] [-S ctl_path]
           [user@]hostname [command]
bash-3.00# id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
bash-3.00# cp /usr/bin/id /usr/bin/ssh
bash-3.00# ssh
A security violation has occurred. Access to the file has been
temporarily disabled. This event has been logged.
bash-3.00# ssh
usage: ssh [-1246AaCfgkMNnqsTtVvXxY] [-b bind_address]
            [-c cipher_spec]
           [-D port] [-e escape_char] [-F configfile]
           [-i identity_file]
           [-L [bind_address:]port:host:hostport]
           [-l login_name] [-m mac_spec] [-O ctl_cmd]
           [-o option] [-p port]
           [-R [bind_address:]port:host:hostport] [-S ctl_path]
           [user@]hostname [command]
bash-3.00#
```

Figure 37 Showing the intrusion remediation system in practice. We can see that once the overwrite attempt has been mitigated, the policy violation logged, the binary is restored.

We see in Figure 37 that the overwrite attempt has been mitigated. The security policy violation has been logged and finally the binary has been restored. We have therefore not even allowed the existence of the malware code on the system. We can chose to move the malware binary to a quarantine location for further analysis.

This concludes the description of the HIPS. The next section shall describe De-perimeterisation. De-perimeterisation is the concept of moving security away from the network perimeter and rather into the applications, operating system and protocols themselves. Our application is one such countermeasure which protects the information asset on which it is installed.

Chapter 7.0 De-perimeterisation

The requirement for real-time data through a company extranet anyplace and anytime and available to all collaborating business partners has created a real-time economy [31].

This requirement for access to real-time data has resulted from concepts very much alike to [40], [52] whereby the authors state that through careful management of information and supply-chain flows amongst collaborating organisations, significant cost reductions in communications overhead, time and error reduction can be achieved while also creating stronger bonds between cross-organisational processes

The rush towards catering and gaining competitive advantage by streamlining business processes so as to achieve real-time organisational informational flows possesses risks [25].

This relationship is portrayed in Figure 38 from [17] showing the relationship between added business value, real-time economy scale and risk against IT systems connectivity and collaborative working.

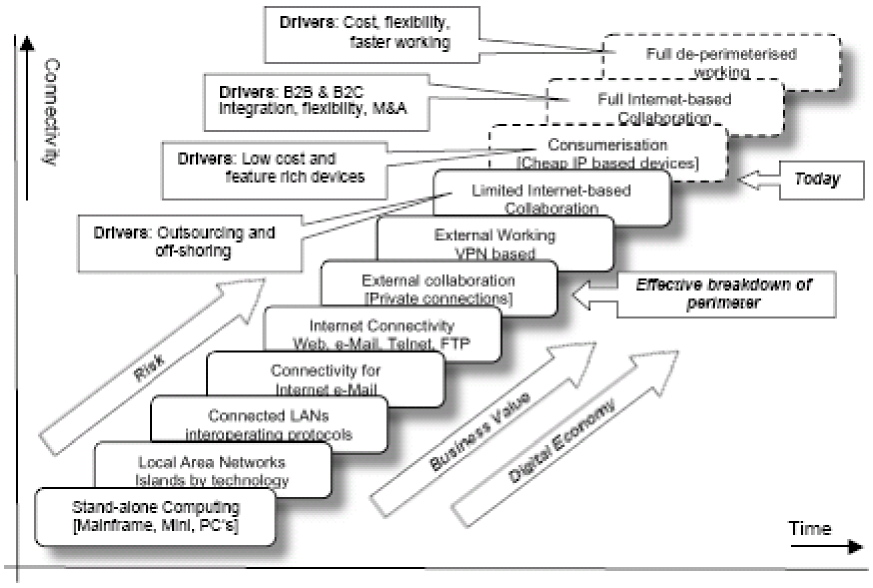

Drivers: Cost, flexibility, faster working
Drivers: B2B & B2C integration, flexibility, M&A
Drivers: Low cost and feature rich devices
Drivers: Outsourcing and off-shoring
Full de-perimeterised working
Full Internet-based Collaboration
Consumerisation [Cheap IP based devices]
Today
Limited Internet-based Collaboration
External Working VPN based
External collaboration [Private connections]
Effective breakdown of perimeter
Internet Connectivity Web, e-Mail, Telnet, FTP
Connectivity for Internet e-Mail
Connected LANs interoperating protocols
Local Area Networks Islands by technology
Stand-alone Computing [Mainframe, Mini, PC's]
Risk
Business Value
Digital Economy
Connectivity
Time

Figure 38 From [17] showing the relationship between added business value, real-time economy scale and risk against IT systems connectivity and collaborative working.

As can be seen in Figure 38, risk exposure increases as an organisation becomes more de-perimeterised. This because external factors *e.g.*, accuracy of information content, stability of information flows and their robustness amidst macro events can all lead to amplified disturbances amongst tranquil business processes ultimately leading to direct loss *i.e.*, monetary loss and delayed losses incurred through loss of reputation.

Within the real-time economy, the mobile worker expects to have access to the same resources they are allowed to use on demand and through whatever type of internet enabled technology they possess at that given point of time [53].

There also exists the need for more granular application-level audit logs. This because the mobile worker shall be interfacing with the corporate IT environment through web services and service oriented architectures.

Hence he will be connecting to trusted systems from un-trusted networks. Modern firewall technology fails to adequately protect information flows against application layer attacks[17], this is primarily attributed to the fact that most firewalls are configured to be porous to web and email based traffic.

Therefore, on one side we have business forces driving towards more open collaboration through protocols and services which flow unobstructed through company firewall architectures and on the other hand, we have the increased business pressures, governance and legislative requirements which explicitly require business to implement effective security controls to minimize risk exposure.

This therefore inevitably adds stress on our rigid perimetric security policy, so much so that effectively it crumbles. Thus the concept of de-perimeterisation is born.

De-perimeterisation according to [39] is a term which was coined to signify the new trend within corporate ICT environments. That is, one of a disappearing reliance on the hardened albeit porous external security perimeter and rather shrinking the perimeter around the core information assets themselves.

It is a concept put forth by the Open Group's Jericho forum which is a industry panel of key market players such as BP, ICI, BT, who have experienced de-perimeterisation happening in their own organizations.

The term can be further be subdivided into macro-perimeterisation *i.e.,* the corporate security perimeter itself and what is known as micro-perimeterisation *i.e.,* moving security away from the perimeter and within the applications being developed, the operating system itself and the protocols being used. Hence achieving a user-centric security model rather than one based upon the hard outside-soft inside model[17].

In [39], [48], a roadmap is proposed for organisations to follow in order to achieve de-perimeterisation. It is composed of five phases. These shall be described in the next section along with how our intrusion protection system is able to address the security requirements possible lapses which exist at each stage.

7.1 Intrusion protection at phase zero

In phase zero, also known as the hard shell model [39] a organisation has a fixed and well defined perimeter. All network access in to and out of the organisation pass through this perimeter. Here, individuals within the organisation are conventionally considered trusted and can thus connect out of the firewall to whichever port and machine they like.

Individuals from the outside are most likely only allowed access to a well defined set of machines sitting within the company's de-militarised zone, DMZ, *i.e.,* a segment of the corporate network considered un-trusted and hence sits between the external boarder firewall and the internal firewall as shown in Figure 39. Within the DMZ, web-servers, email servers are most likely to be found.

External individuals requiring access to corporate internal servers are most likely going to interface with them using virtual private networking, VPN, software. VPN software establishes IPSec or SSL/TLS tunnels and authenticates the client to the server, giving the client an internal IP and hence access to protected resources within the corporate LAN.

Phase zero describes the IT architecture that most networks are modelled by today [39].

The main security lapse here is that internal systems are not protected from attacks originating from behind the firewalls[14]. Furthermore, as the firewalls are porous to web and email and other application traffic, then such attacks are allowed to go through and many times successfully exploit machines within the DMZ.

One might at this stage argue that a compromised machine within the DMZ poses very little danger of being able to attack internal LAN machines. This because there exists a firewall as shown in Figure 39 stopping the DMZ machines from connecting to the LAN.

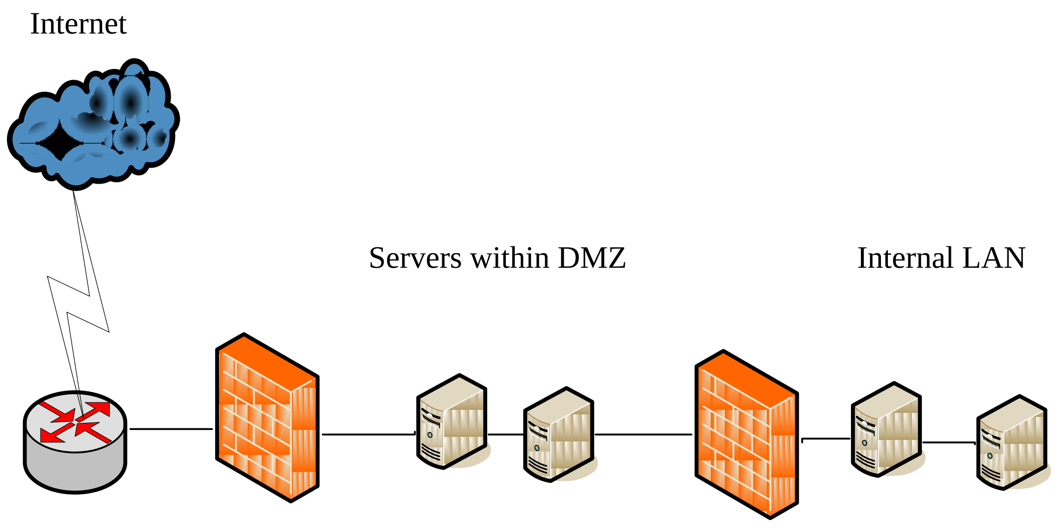


Figure 39 Showing a typical DMZ setup.

However this is not true in practice, an attacker can easily mount a web browser client-side exploit which shall attack all machines connecting from the internal LAN on to the web-server within the DMZ.

What has to be done on the side of the internal LAN user is to simply browse to the page hosting the attack code which shall exploit the client's browser sending a reverse console shell to the attacker *i.e.*, from the internal LAN machine to the DMZ compromised server hosting the exploit page.

The compromised server within the DMZ has now been able to compromise an internal LAN machine despite the firewall barrier which would leave us to believe that internal LAN machines are safe from such attacks. Clearly not the case.

Our NIPS system is able to protect the corporate LAN against the attack described above. We demonstrated this in section 6.1.1.5.

In this case when the internal LAN client would browse to the exploit containing web-site on the DMZ web-server, the NIPS running on the internal LAN server would detect shellcode within the OSI TCP/IP application layer.

The NIPS would then invoke *iptables* which would insert the attackers IP within its INPUT and OUTBOUND deny rules. Hence any reverse connect attempts back to the compromised DMZ server are thwarted.

7.2 Intrusion protection at phase one

Phase one according to [48] involves moving non-corporate items outside the corporate perimeter combined with the deliverance of external services outside the corporate perimeter.

This stage according to [39] is the first stage of de-perimeterisation and primarily involves establishing the enabling technology for remote working of employees and collaborating third parities.

The key security issue here has been identified [48] as the secure use of non-secure computers.

Here our IPS is able to offer a solution. Firstly as discussed in section 6.1.2, the HIPS is able to verify on run time all executed binaries. Hence at any given point in time, one gains assurance that the programs being run by the user on the local machine are in a verified and secure state and thus have not been compromised.

Secondly the NIPS monitors all incoming and outgoing TCP/IP traffic for malicious payloads as discussed in section 6.1.1. We achieve both outgoing and incoming packet payload inspecting by hooking the NF_IP_POST_ROUTING Netfileter hook which parses all packets either destined for our host, traversing it, or leaving it.

We are able to thwart network centric attacks when they are detected as discussed in 6.1.1.5.

Therefore by installing the IPS on our workers personal computers and our company servers, we can gain a reasonable amount of assurance that the computers are operating within a verified state and that any attempt to propagate exploit code will be stopped.

Whilst at the same time, due to the significantly lower overheads that the system posses as opposed to similar approaches we described in Chapter 5.0, the IPS will not impede normal PC operation, both computationally and network processing wise.

Therefore, our IPS satisfies the requirements for protection at phase one by providing assurance during the use of non-secure computers.

7.3 Intrusion protection at phase two

In phase two according to [48] we remove the hardened perimeter, it becomes a quality of service, QoS, boundary. Paul Simmonds goes on to say that essentially the firewall shall become a sieve able to separate lumps *i.e.*, it shall merely be present not for security reasons but as a means of network optimisation.

Critical business systems shall be made directly available on the internet. As will systems belonging to the organisations partners. The security of these critical business systems in phase two shall rely more so on the use of strong authentication, encryption and the use of type-safe languages such as Java[39] than the existence of a security perimeter- firewall.

In phase two one of the critical security issues that were identified by Paul Simmonds includes the ability to monitor and correlate security information from many sources.

Our IPS has been developed with distributed logging in mind. Including this functionality is a matter of logging to a central database all security related violations from all IPS sensors distributed within the LAN.

This includes creating a logging program for our NIPS which shall be executed by *call_usermodehelper* with arguments the machine name, the source IP of the attacker, the signature of the attack and time stamp. This is shown in Figure 40.

```
mysql> describe nips;
+------------------+--------------+------+-----+------------+
| Field            | Type         | Null | Key | Default    |
+------------------+--------------+------+-----+------------+
| Id               | int(10)      |      | PRI | NULL       |
| Host             | varchar(50)  |      |     |            |
| Source_IP        | varchar(50)  |      |     |            |
| Attack_Signature | varchar(250) |      |     |            |
| Criticality      | varchar(50)  |      |     |            |
| Time             | timestamp    |      |     | CURRENT_TIM|
| Date             | date         |      |     | 0000-00-00 |
+------------------+--------------+------+-----+-----+------+
7 rows in set (0.00 sec)
mysql>
```

Figure 40 MySQL database table for the network intrusion protection system. Stores attack event information.

Having stored this security violation info within our database table, correlating which events happened where and the extent of each attack is a simple matter of executing the relevant SQL query for the information being sought.

For example, to display all critical security events that have been aggregated within the database from all IPS sensors:

```
mysql> select host,source_ip,attack_signature,date from nips where
criticality='critical';
+--------------+-------------+------------------+------------+
| host         | source_ip   | attack_signature | date       |
+--------------+-------------+------------------+------------+
| web-server   | 192.168.0.1 | sys_exit         | 2007-08-15 |
| email-server | 192.168.0.1 | sys_exit         | 2007-08-15 |
+--------------+-------------+------------------+------------+
2 rows in set (0.00 sec)

mysql>
```

Figure 41 Issuing simple SQL queries against the NIPS database to determine which critical security events have been detected across the LAN.

Due to rendering of the security perimeter to a simple QoS boundary in phase two [39], the applications which are chosen to be placed outside of the boundary must be inherently secure. Any window of opportunity for exploitation shall give a potential attacker lots of power to mount very effective attacks.

As these are externally accessible business critical systems through which company workers either within the premises or working from remote access the risk of an attacker mounting client-side browser exploits still lingers.

For the same reasons given before when describing intrusion protection in phase one against client-side exploits, our NIPS is able to protect these business critical applications despite any inherent software vulnerabilities which may be present in the business critical applications themselves.

Effectively the IPS system at phase two, paraphrasing Steve Bellovin[4], is able to act as a network security solution for a software engineering problem.

7.4 Intrusion protection at phase three

In phase three Simmonds asserts that we must move from system level authentication to data level validation and connection authentication. He proposes a type of role based data authentication mechanism whereby one authenticates to the data he is permitted to view[48].

The types of enabling technologies which Simmonds proposes include databases with table and cell encryption. As well as cross business authentication of credentials[48].

He also proposes that application level firewalls and IDS systems will be enabling technologies for phase three. Hence at phase three what we see is move in firewall detection from layer three to layer seven in the OSI TCP/IP layer stack *i.e.,* the application layer. That goes to say that such network protection devices shall become more TCP/IP payload content aware.

The IPS system as a whole is one such application-layer aware system as discussed in section 6.1.1. It is able to protect applications running on the system on which it is installed from network application layer attacks by matching TCP/IP application layer data with known shellcode signatures 6.1.1.4.

According to [39] the firewall/IDS/IPS systems operating at phase three will have to intercept, parse and protect from application layer attacks at a scale not seen previously.

In Chapter 5.0 we discussed that the NIPS being kernel based avoids continuous context switches into userland to evaluate the packets payloads. This is done entirely in kernel land.

We do however perform a context switch to user-land when we invoke *iptables* however this is only ever done when a security violation has occurred; the actual filtering and signature matching occurs entirely in kernel land.

Unlike the approach used by Sourcefire’s Snort which invokes a context switch whenever a packet is met on the wire, we try to keep such context switches down to the minimum.

Therefore this approach is much faster and hence offers this advantage that we are seeking for phase three, in speed of packet processing at scales not seen in previous approaches.

Therefore, the IPS is able to comply with the firewall/IDS/IPS requirements set forth in phase three.

7.5 Intrusion protection at phase four

Phase four involves what Simmonds calls [48] data level authentication. The perimeter has been completely abolished in the previous stages and now commercial IT environments are operating within a boundary-less information flow model[24],[39].

Boundary-less information flow is an information flow model realised by the business requirement for integrated company, supplier and customer information and access to that information. It is heavily reliant on the use of web-based technology as a means of refining business processes to achieve competitive edge and real-time access to information [24].

Security in phase four according to [39] is completely integrated. Simmonds gives an example of a simple file transfer. Today when such an operation is done from one machine to another, the file copied over is created with the security and read, write, execute permissions of the user performing the operation locally.

In phase four, such an operation will result in the file being copied over with its security information maintained *i.e.,* the same as the original file possessed[48].

This specific example is not something that our IPS is currently able to achieve to the same extent without taking some large assumptions and without modification.

The HIPS is able to perform binary run-time verification. Provided the MD5 hash database is the same on both machines which would mean that the compile-time environment would be homogeneous as would the system specifications. Then, when one binary is transferred to another machine, its run-time binary verification would be successful.

However, to maintain read, write execute access during cross-machine transfers one would have to encode the permissions along side with the MD5 hash of the binary within the hash database. So as when the binary is transferred from machine to machine, then the HIPS is able to re-instate the permissions of the binary.

This functionality is not something currently available within the IPS, however given the time frames of when phase four is likely to take place according to under predictions made in [48] *i.e.*, 2008 onwards, the IPS can currently address IPS requirements at phases zero to three and on-going work would ensure that such integrated functionality would be ready when businesses start addressing phase four de-perimeterisation.

Under prediction because for phase four to be realised according to [39] all public computing infrastructures will have to be upgraded to "make them harmonized validated secure environments". This in its very self hides lots of effort as it shall require new standards to address such factors such as data classification, cross company authentication and the existence of "open validated secure environments"[48].

One of the main business issues hampering wide-spread adoption of the boundary-less information flow model according to [24] is security *i.e.,* protecting the integrity and confidentiality of the information in question as it is accessed and used.

As the technology behind boundarly-less information flows itself is highly reliant on web-based applications our NIPS as previously discussed would be able to protect the web-servers running the applications from attack. Therefore, serving as an impetus for more confidence in adoption of de-perimeterisation.

We have come to the conclusion that our IPS system as a whole is able to address the firewall/IDS specific issues that exist within phases zero to three of the Open Group's Jericho Forum's proposal for de-perimeterisation. Once further enabling-technology developments occur then it shall also be able to address phase four de-perimeterisation issues to a greater extent.

Chapter 7.0 Conclusion

7.1 Summary

The objective of this dissertation was to develop a Linux host and network intrusion protection system which shall be suitable for deployment in de-perimeterised networks.

The NIPS was required to be able to operate within high-speed networks. This because phase three of the Jericho forum's proposal for de-perimeterisation [39],[48] requires that new firewall/IDS/IPS technologies intercept packets at speeds not seen previously.

To achieve this goal, the NIPS was designed to intercept packets and perform its signature matching entirely within the Linux kernel. Hence avoiding performance costly context switches to user land. Resorting only to performing a context switch when adding an attackers IP within the firewall deny rules.

It therefore offers distinct performance advantages over similar solutions mostly based on libpcap *e.g.*, Sourcefire's Snort, which resort to copying all packets over to user land where they are then processed.

Phase three of the Open Group's Jericho forum for de-perimeterisation also requires that current firewall/IDS/IPS technologies shift from layer three to layer seven of the TCP/IP processing. This was also therefore one of the key requirements for our NIPS *i.e.*, to be application layer aware.

The NIPS was designed to perform signature matching against application-payer payloads. It therefore satisfies this requirement of the Jericho forum as it is able to actively protect against application-layer attacks as we demonstrated in Figure 30 and Figure 31.

One further requirement is the ability to monitor and correlate security information from many sources. Here once more the IPS system is able to be configured to store its security event information within a MySQL database, from which one can correlate security events which happen across the LAN as described in section 7.3.

The HIPS addresses the issues surrounding the secure use of non-secure computers described in phase one of the Jericho forum's de-perimeteristaion requirements. At any given point in time, one gains assurance that the programs being run by the user on the local machine are in a verified and secure state and thus have not been compromised.

The HIPS offers a higher degree of fault tolerance due to the fact that, unlike Tripwire, the we have described how the HIPS is able to prevent malicious modification of core operating system binaries which could compromise integrity, availability and confidentiality.

Furthermore, we introduced the concept of intrusion remediation systems whose purpose is to reinstate a system back into a known good state of operation post-intrusion.

The dissertation therefore addressed the goals it set out to achieve.

7.2 Further work

Chapter 8.0 References

1. *Sarbanes Oxley Act* of 2002 at http://frwebgate.access.gpo.gov/cgi-bin/getdoc.cgi?dbname=107_cong_bills&docid=f:h3763enr.tst.pdf
2. Alfonso Valdes, M.A., Steven Cheung, Yves Dutertre, Victoria Stavridou, Tomas E. Uribe, Joshua Levy, Hassen Sadi, An architecture for an adaptive intrusion tolerant server. In *Proceeds of the 10th international workshop on security protocols*, (Cambridge, UK, 2002).
3. Axelsson, S. *Intrusion Detection Systems: A Survey and Taxonomy*, Chalmers University of Technology, 2000 at http://www.cs.plu.edu/courses/CompSec/arts/taxonomy.pdf
4. Bellovin, S. *Shifting the Odds, Writing More Secure Software*, AT&T Research, 1996 at http://www.cs.columbia.edu/~smb/talks/odds.pdf
5. Benjamin A. Kuperman, E.S. *Generation of Application Level Audit Data via Library Interposition (Technical Report)*, Purdue University, COAST Laboratory, 1999 at https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/99-11.pdf
6. Benvenuti, C. *Understanding Linux Network Internals*. O'Reilly, California, 2005.
7. Bioforge. Hacking the Linux Kernel Network Stack, *Phrack Magazine*, 2003 at http://www.phrack.org/issues.html?issue=61&id=13&mode=txt
8. C Abad, J.T., C Sengul, W Yurcik, Log correlation for intrusion detection: a proof of concept. In *Computer Security Applications Conference*, (Las Vegas, 2003).
9. C. Kruegel, D.M., W. Robertson, G. Vigna, R. Kemmerer, Reverse Engineering Network Signatures. In *Proceedings of the AusCERT Asia Pacific Information Technology Security Conference*, (Gold Coast, Australia, 2005).
10. C. Taylor, A.W.K., N. Hanebutte, M. McQueen, Low-Level Network Attack Recognition: A Signature-Based Approach. In *IEEE Proceedings of the 14th International Conference on Parallel and Distributed Computing Systems*, (Dallas, 2001).
11. CERT. *Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks*, Carnegie Mellon University, 1996 at http://www.cert.org/advisories/CA-1996-21.html
12. CERT. *Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests*, Carnegie Mellon University, 2000 at http://www.cert.org/advisories/CA-2000-02.html
13. Chen-Hwa Song, Y.-Y.H. Detecting method and architecture thereof for malicious codes *US2006/0143707 A1*, United States, 2005.
14. Chris Payne, T.M., Architecture and Applciations for a Distributed Embedded Firewall. In *17th Annual Computer Security Applications Conference*, (Louisiana, USA, 2001).
15. Curtis Carver, J.H., Udo Pooch, Limiting Uncertainty in Intrusion Response. In *Proceedings of the IEEE Workshop on Information Assurance and Security United States Military Academy*, (New York, 2001).
16. David Moore, C.S., K Claffy, Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Internet Measurement Conference*.

Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement, (New York, USA, 2002).

17. Dobson, I. *Business Case for De-perimeterisation (White Paper)*, The Jericho Forum, 2007 at http://www.opengroup.org/jericho/Business_Case_for_DP_v1.0.pdf
18. Edge, J., Remote file inclusion vulnerabilities, 2006, http://lwn.net/Articles/203904/
19. Eugene Spafford, G.K., The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, (New York, USA, 1994).
20. Faust, S. *LDAP Injection Are your web applications vulnerable?*, 2005 at http://www.spidynamics.com/whitepapers/LDAPinjection.pdf
21. Fulvio Risso, L.D., An Architecture for High Performance Network Analysis. In *Sixth IEEE Symposium on Computers and Communications*, (Tunisia, 2001).
22. Hannu Kari, C.C., Jane Lundberg, Packet Level Authentication in Military Networks. In *Proceedings of the 6th Australian Information Warfare & IT Security Conference*, (Geelong, Australia, 2005).
23. Harris, S. *CISSP All In One*. McGraw-Hill, California, 2005.
24. Holmes, P. *An Introduction to Boundaryless Information Flow*, The Open Group, 2002 at http://www.opengroup.org/bookstore/catalog/w201.htm
25. Hope, W. Global Capitalism and the Critique of Real Time. *Time & Society*, *15* (2). 275-302.
26. Intel. Basic Architecture. in *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Denver, 2007.
27. Izik, Reverse engineering with LD_PRELOAD, http://securityvulns.com/articles/reveng/
28. Jack Koziol, D.L., Dave Aitel *The Shellcoders Handbook*. Wiley, Indianapolis, 2004.
29. John Rochlis, M.E., With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, (California, 1989).
30. Jonathan Corbet, A.R., Greg Kroah-Hartman *Linux Device Drivers Third Edition*. O'Reilly Media, 2005.
31. Jurgen Bohn, V.C., Marc Langheinrich, Friedemann Mattern, Michael Rohs Living in a World of Smart Everyday Objects - Social, Economic, and Ethical Implications. *Human and Ecological Risk Assessment*, *10* (5). 763-785.
32. Kong, J. *Designing BSD Rootkits*. No Starch Press, San Francisco, 2007.
33. Mark Crosbie, E.S., Defending a Computer System using Autonomous Agents. In *8th National Information Systems Security Conference*, (Virginia, 1994).
34. McAfee. *Top ten requirements for next-generation IDS (White Paper)*, 2003 at http://www.mcafee.com/us/local_content/white_papers/wp_intruverttop10.pdf
35. McCarthy, J., Debian Project Servers Compromised, 2003, http://linux.slashdot.org/article.pl?sid=03/11/21/1314238
36. Miller, T., T0rn rootkit analysis, 2005, http://www.ossec.net/rootkits/studies/t0rn.txt

37. Nenad Jovanovic, E.K., Christopher Kruegel. *Preventing Cross Site Request Forgery Attacks (Technical Report)*, Technial University of Vienna, 2007 at http://www.seclab.tuwien.ac.at/papers/noforge_techreport.pdf
38. One, A. Smashing the Stack for Fun And Profit, *Phrack Magazine*, 1996 at http://www.phrack.org/archives/49/P49-14
39. Palmer, G. De-Perimeterisation: Benefits and limitations. *Information Security Technical Report*, *10*. 189-203.
40. Qizhi Dai, R.K., Business Models for Internet-based E-Procurement Systems and B2B Electronic Markets: An Exploratory Assessment. In *34th Hawaii Internationa Clonference on Systems Science*, (Maui, Hawaii, 2000).
41. Randy Russel, H.W., Linux Netfilter Hacking Howto, 2002, http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html
42. Rivest, R.L. *RFC 1321. The MD5 message-digest algorithm (Technical Report)*, 1992 at http://www.faqs.org/rfcs/rfc1321.html
43. Robin Sommer, V.P., Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Conference on Computer and Communications Security. Proceedings of the 10th ACM Conference on Computer and Communications Security*, (New York, USA, 2003).
44. Roesch, M. *Snort: Lightweight Intrusion Detection for Networks*, 1999 at http://www.snort.org
45. Russel Bradford, R.S., Brian Unger, Packet Reading for Network Emulation. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, (Ohio, USA, 2001), IEEE Computer Society.
46. Sandeep Kumar, E.S. *An Application of Pattern Matching in Intrusion Detection (Technical Report)*, Purdue University, 1994 at http://citeseer.ist.psu.edu/389890.html
47. SANS. *Top-20 Internet Security Attack Targets*, 2006 at http://www.sans.org/top20/
48. Simmonds, P., Deperimeterisation: This Decade's Security Challenge. In *Blackhat Asia Conference 2004*, (Japan, 2004).
49. Spett, K. *SQL Injection Are your web applications vulnerable?*, 2005 at http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf
50. Symantec. *Symantec Internet Security Threat Report*, 2007 at http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_internet_security_threat_report_xi_03_2007.en-us.pdf
51. Tatyana Ryotov, C.N., Dongho Kim, Li Zhou Integrated Access Control and Intrusion Detection for Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, *14* (9). 841-850.
52. Thomas Malone, J.Y., Robert Benjamin Electronic Markets and Electronic Hierarchies. *Communications of the ACM*, *30* (6). 484-497.
53. Tomlinson, G. Open on all sides, he cannot be attacked, *Infosecurity Today*, 2004 at http://dx.doi.org/10.1016/S1742-6847(04)00069-2
54. Toshihiro Tabata, K.S., Design of Intrusion Detection System at User Level with System-Call Interposing. In *1st International Conference on E-Business and Telecommunication Networks*, (Portugal, 2004).
55. Van Jacobson, C.L., Steven McCanne. libpcap, Lawrence Berkeley National Laboratory, 1994.

56. W Lee, S.S., P Chan, E Eskin, W fan, Real time data mining-based intrusion detection. In *DARPA Information Survivability Conference & Exposition II*, (California, 2001).
57. Wenke Lee, J.B.D.C., Ashley Thomas, Niranjan Balwalli, Sunmeet Saluja, Yi Zhang, Performance Adaption in Real-Time Intrusion Detection Systems. In *Recent Advances in Intrusion Detection : 5th International Symposium*, (Zurich, Switzerland, 2002).
58. Wolthusen, S.D., Molehunt: Near-line Semantic Activity Tracing. In *Proceedings of the 2004 IEEE Workshop on Information Assurance and Security*, (New York, 2004).
59. Wolthusen, S.D. Self-Inflicted Vulnerabilities. in *Naval War College Review*, 2004, 102-113.
60. Xiaoyun Wang, H.Y. How to Break MD5 and Other Hash Functions. in *Advances in Cryptology - EUROCRYPT*, 2005, 19-35.
61. Z Liang, R., Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *Conference on Computer and Communications Security. Proceedings of the 12th ACM conference on Computer and communications security*, (Virginia, 2005).

Chapter 9.0 Glossary

A
B
C
D
E
F
G
H

Hook

According to [32] a hook is a programming method which uses handler functions (*hooks*) which modify the flow of execution. A new hook registers its address as the location for a specific function, so that when the function is called, the hook is executed instead.

I
J
K
L
M
N
O
P
Polymorphic
Shellcode which has been purposely obfuscated for the sole reason of IDS evasion.

Q
R
S

Segmentation Fault
A segmentation fault (sometimes referred to as segfault for short) is a particular error condition that can occur during the operation of computer software. In short, a segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way it is not allowed to (eg, attempts to write a read-only location)

SYN flood
A SYN flood is a Denial of Service attack which takes advantage of the three-way TCP/IP handshake. It involves sending many SYN packets to an open TCP port on the machine. This exhausts resources, denying legitimate users from accessing the service running on that port. More information can be found in [11].

T
U
V
W

X

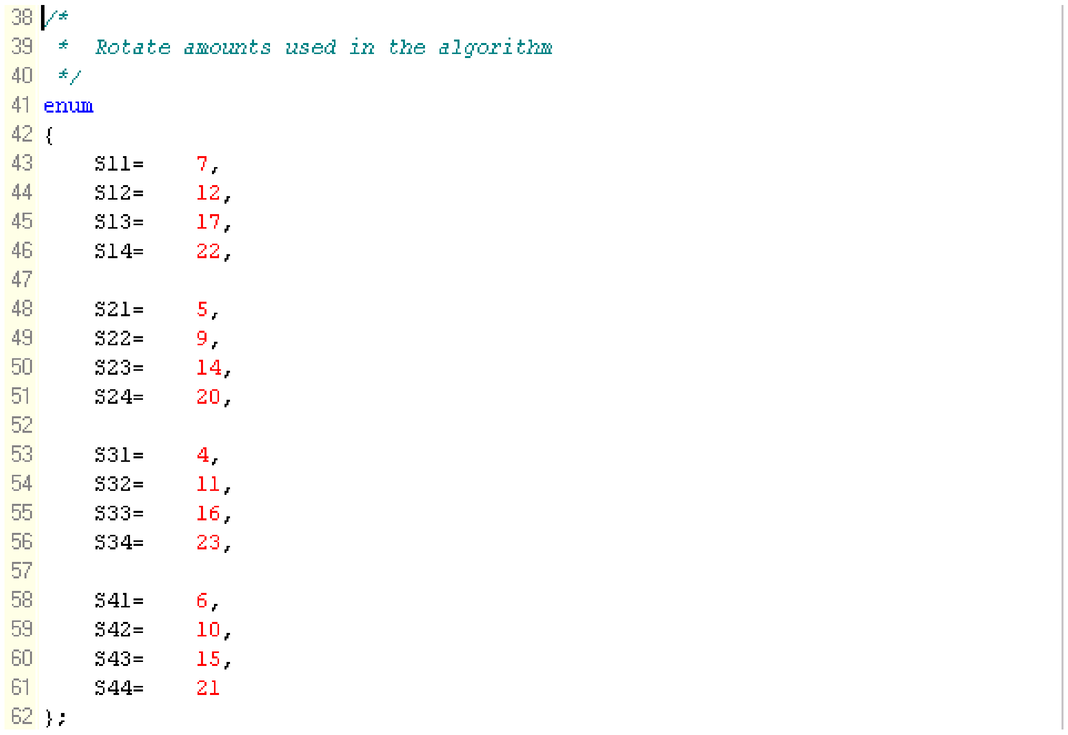
Y

Z

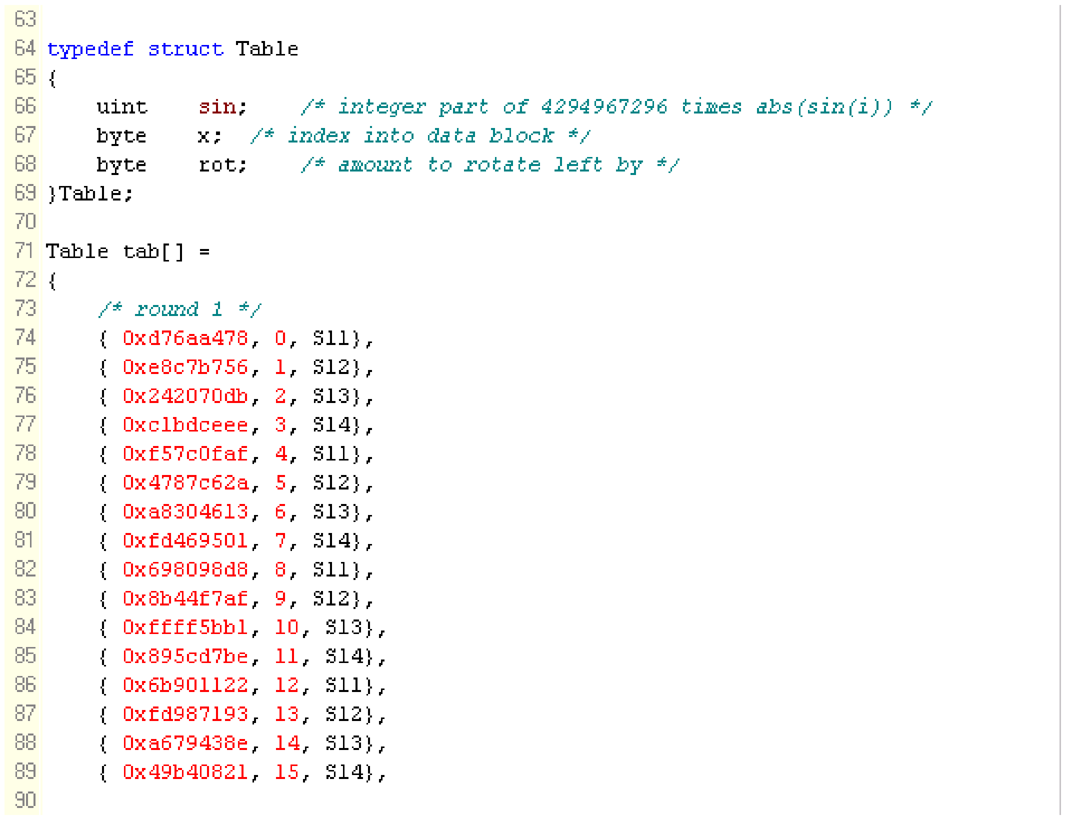
Chapter 10.0 Appendices

10.1 Host Intrusion Protection System

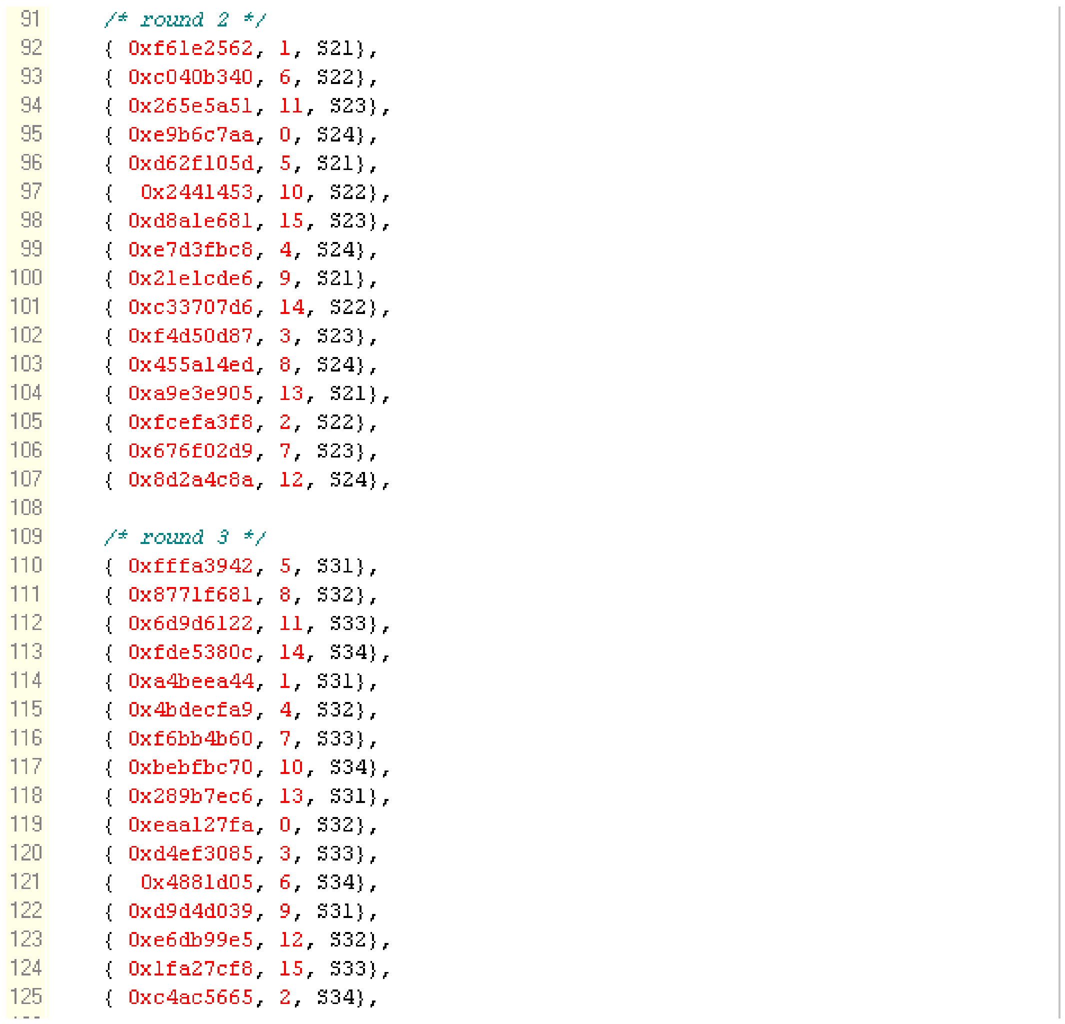
```
/*
                Christian Papathanasiou

Information Security Group, Royal Holloway University of London

            Run-time binary integrity checker

    gcc -fPIC -shared ld-preload-redirect.c -o ld-preload.so
    mv ld-preload.so /lib/
    echo "/lib/ld-preload.so">>/etc/ld.so.preload
*/


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <dlfcn.h>

#define lib_path "/lib/libc.so.6"


typedef unsigned int uint;
typedef unsigned char byte;
extern int enc64(char*,byte*,int);



/*
    The below MD5 hashing routines are very slightly modified
   from http://www.netlib.org/crc/md5sum.c and are
   Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991.
*/
```



```c
/*
 *  Rotate amounts used in the algorithm
 */
enum
{
    S11=    7,
    S12=    12,
    S13=    17,
    S14=    22,

    S21=    5,
    S22=    9,
    S23=    14,
    S24=    20,

    S31=    4,
    S32=    11,
    S33=    16,
    S34=    23,

    S41=    6,
    S42=    10,
    S43=    15,
    S44=    21
};
```

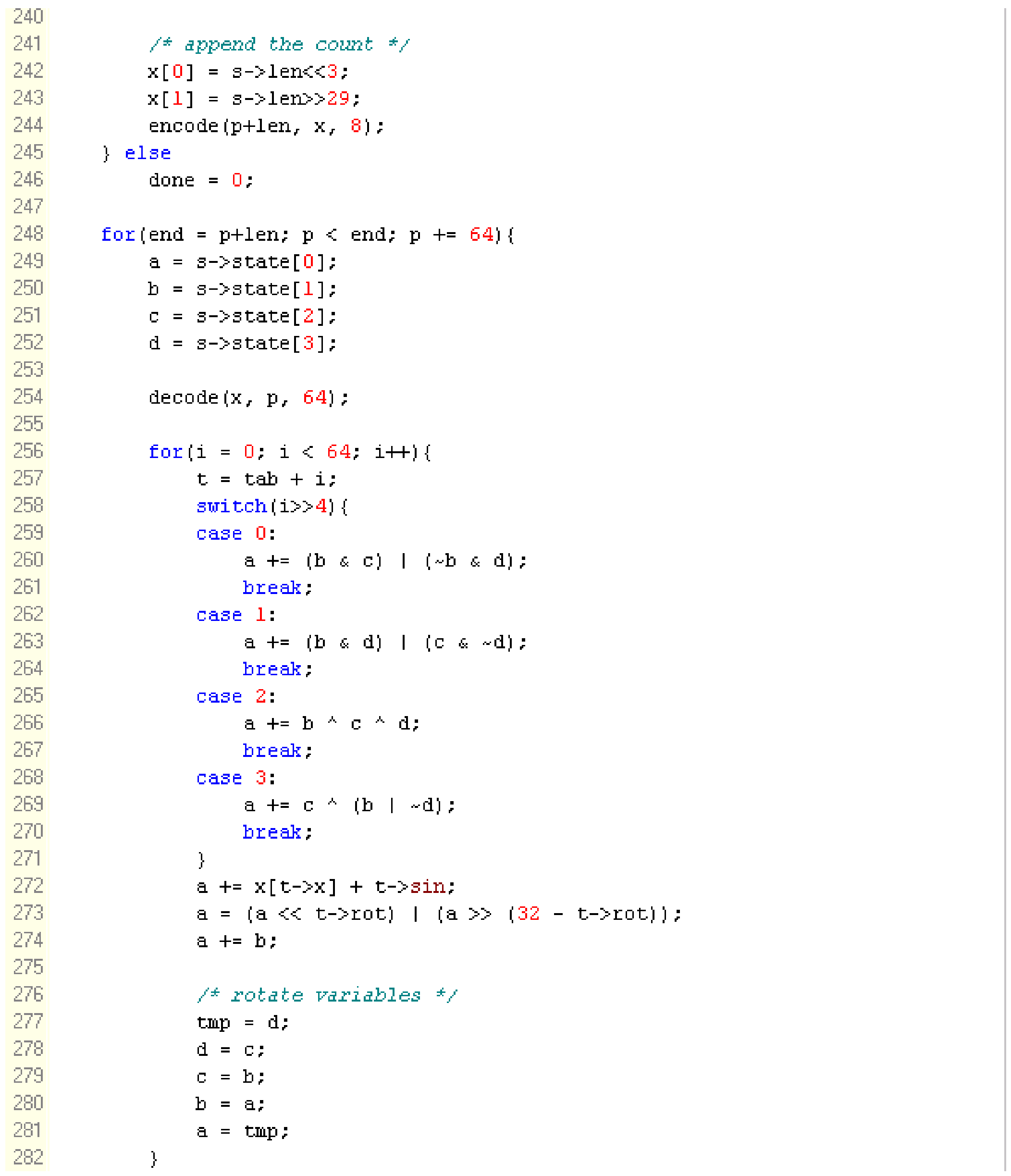


```c

typedef struct Table
{
    uint    sin;    /* integer part of 4294967296 times abs(sin(i)) */
    byte    x;  /* index into data block */
    byte    rot;    /* amount to rotate left by */
}Table;

Table tab[] =
{
    /* round 1 */
    { 0xd76aa478, 0, S11},
    { 0xe8c7b756, 1, S12},
    { 0x242070db, 2, S13},
    { 0xc1bdceee, 3, S14},
    { 0xf57c0faf, 4, S11},
    { 0x4787c62a, 5, S12},
    { 0xa8304613, 6, S13},
    { 0xfd469501, 7, S14},
    { 0x698098d8, 8, S11},
    { 0x8b44f7af, 9, S12},
    { 0xffff5bb1, 10, S13},
    { 0x895cd7be, 11, S14},
    { 0x6b901122, 12, S11},
    { 0xfd987193, 13, S12},
    { 0xa679438e, 14, S13},
    { 0x49b40821, 15, S14},

```



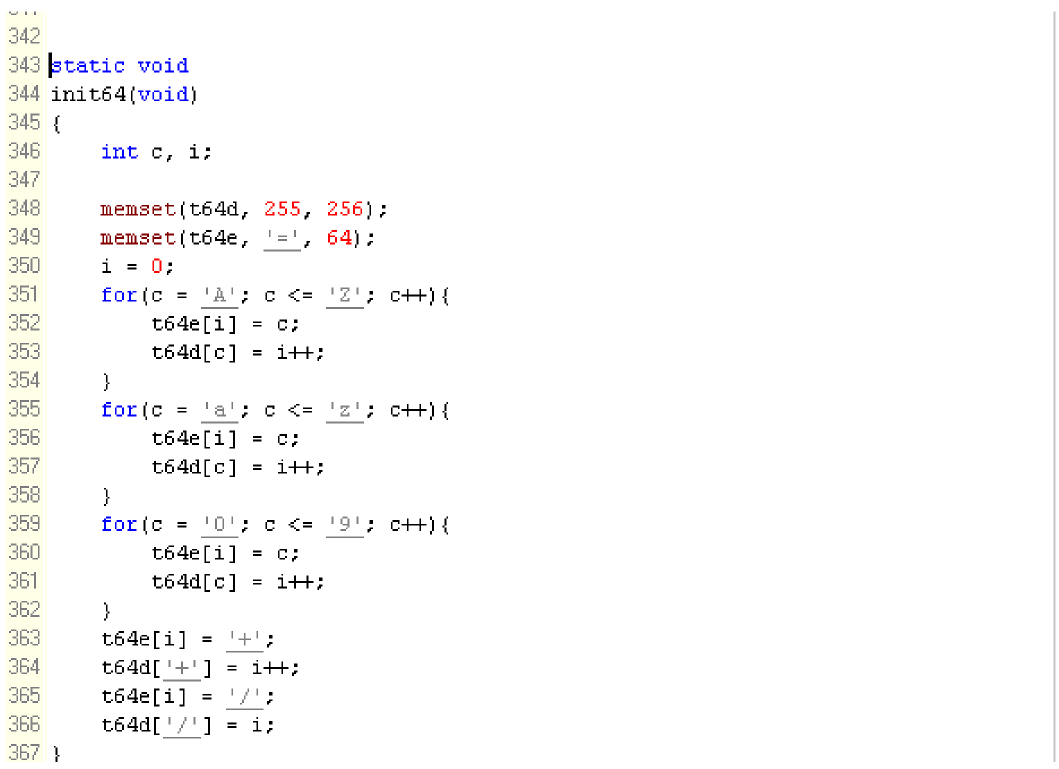
```
    /* round 2 */
    { 0xf61e2562, 1, S21},
    { 0xc040b340, 6, S22},
    { 0x265e5a51, 11, S23},
    { 0xe9b6c7aa, 0, S24},
    { 0xd62f105d, 5, S21},
    {  0x2441453, 10, S22},
    { 0xd8a1e681, 15, S23},
    { 0xe7d3fbc8, 4, S24},
    { 0x21e1cde6, 9, S21},
    { 0xc33707d6, 14, S22},
    { 0xf4d50d87, 3, S23},
    { 0x455a14ed, 8, S24},
    { 0xa9e3e905, 13, S21},
    { 0xfcefa3f8, 2, S22},
    { 0x676f02d9, 7, S23},
    { 0x8d2a4c8a, 12, S24},

    /* round 3 */
    { 0xfffa3942, 5, S31},
    { 0x8771f681, 8, S32},
    { 0x6d9d6122, 11, S33},
    { 0xfde5380c, 14, S34},
    { 0xa4beea44, 1, S31},
    { 0x4bdecfa9, 4, S32},
    { 0xf6bb4b60, 7, S33},
    { 0xbebfbc70, 10, S34},
    { 0x289b7ec6, 13, S31},
    { 0xeaa127fa, 0, S32},
    { 0xd4ef3085, 3, S33},
    {  0x4881d05, 6, S34},
    { 0xd9d4d039, 9, S31},
    { 0xe6db99e5, 12, S32},
    { 0x1fa27cf8, 15, S33},
    { 0xc4ac5665, 2, S34},

    /* round 4 */
    { 0xf4292244, 0, S41},
    { 0x432aff97, 7, S42},
    { 0xab9423a7, 14, S43},
    { 0xfc93a039, 5, S44},
    { 0x655b59c3, 12, S41},
    { 0x8f0ccc92, 3, S42},
    { 0xffeff47d, 10, S43},
    { 0x85845dd1, 1, S44},
    { 0x6fa87e4f, 8, S41},
    { 0xfe2ce6e0, 15, S42},
    { 0xa3014314, 6, S43},
    { 0x4e0811a1, 13, S44},
    { 0xf7537e82, 4, S41},
    { 0xbd3af235, 11, S42},
    { 0x2ad7d2bb, 2, S43},
    { 0xeb86d391, 9, S44},
};

typedef struct MD5state
{
    uint len;
    uint state[4];
}MD5state;
MD5state *nil;

int debug;
int hex;    /* print in hex?  (instead of default base64) */

void encode(byte*, uint*, uint);
void decode(uint*, byte*, uint);
MD5state* md5(byte*, uint, byte*, MD5state*);
char * sum(FILE*, char*);

char *
sum(FILE *fd, char *name)
{
    byte *buf;
    byte digest[16];
    char pr64[25];

    char md5buf[100];

    int i, n;
    MD5state *s;
    s = nil;
    n = 0;
    buf = calloc(256,64);
    for(;;){
        i = fread(buf+n, 1, 128*64-n, fd);
        if(i <= 0)
            break;
        n += i;
        if(n & 0x3f)
            continue;
        s = md5(buf, n, 0, s);
        n = 0;
    }
    md5(buf, n, digest, s);

        enc64(pr64,digest,sizeof(digest));
        pr64[22] = '\0';  /* chop trailing == */


    snprintf(md5buf,sizeof(md5buf),"%s",pr64);
    return md5buf;

    free(buf);

}
```

```c



/*
 *  I require len to be a multiple of 64 for all but
 *  the last call
 */
MD5state*
md5(byte *p, uint len, byte *digest, MD5state *s)
{
    uint a, b, c, d, tmp;
    uint i, done;
    Table *t;
    byte *end;
    uint x[16];

    if(s == nil){
        s = calloc(sizeof(*s),1);
        if(s == nil)
            return nil;

        /* seed the state, these constants would look nicer big-endian */
        s->state[0] = 0x67452301;
        s->state[1] = 0xefcdab89;
        s->state[2] = 0x98badcfe;
        s->state[3] = 0x10325476;
    }
    s->len += len;

    i = len & 0x3f;
    if(i || len == 0){
        done = 1;

        /* pad the input, assume there's room */
        if(i < 56)
            i = 56 - i;
        else
            i = 120 - i;
        if(i > 0){
            memset(p + len, 0, i);
            p[len] = 0x80;
        }
        len += i;
```

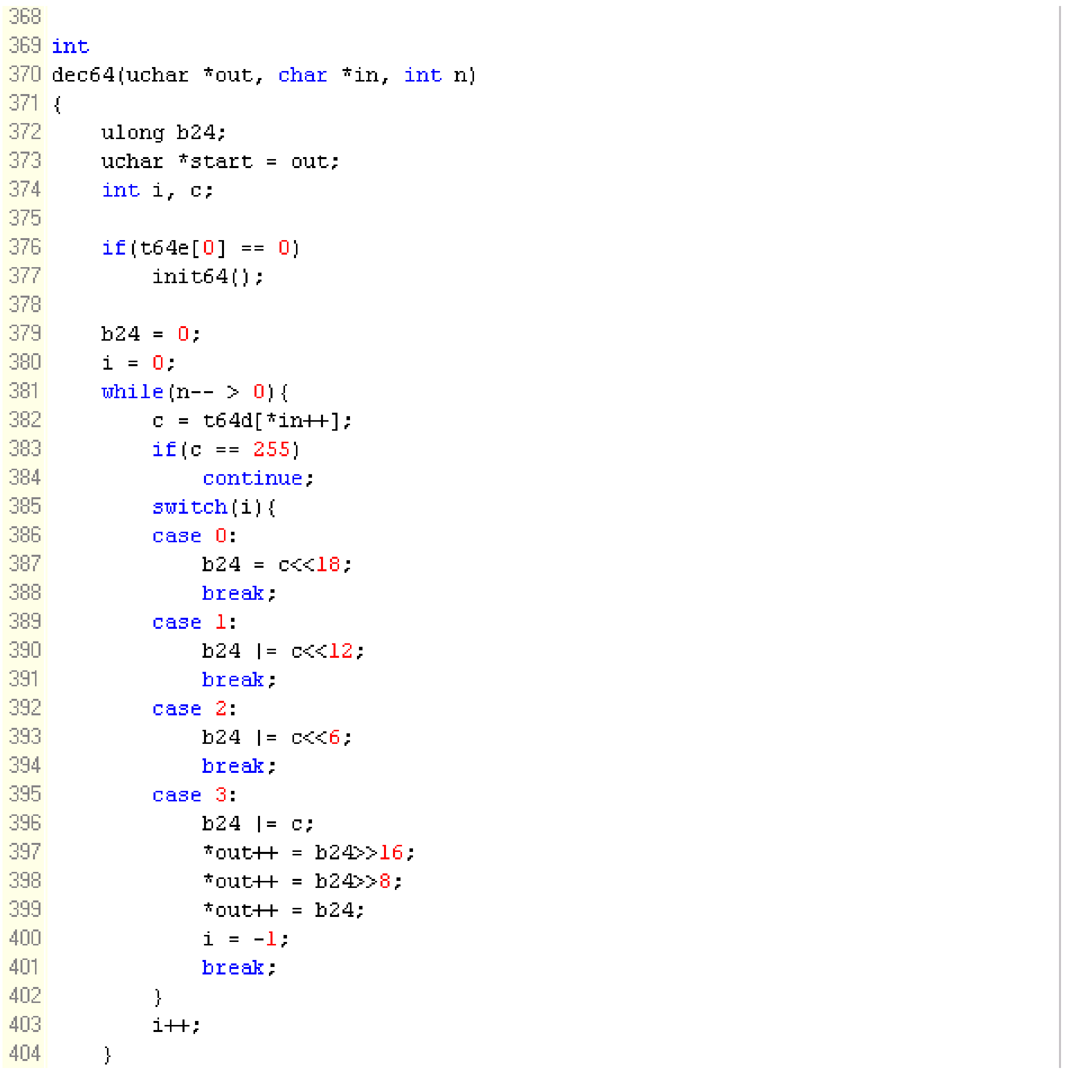


```c

        /* append the count */
        x[0] = s->len<<3;
        x[1] = s->len>>29;
        encode(p+len, x, 8);
    } else
        done = 0;

    for(end = p+len; p < end; p += 64){
        a = s->state[0];
        b = s->state[1];
        c = s->state[2];
        d = s->state[3];

        decode(x, p, 64);

        for(i = 0; i < 64; i++){
            t = tab + i;
            switch(i>>4){
            case 0:
                a += (b & c) | (~b & d);
                break;
            case 1:
                a += (b & d) | (c & ~d);
                break;
            case 2:
                a += b ^ c ^ d;
                break;
            case 3:
                a += c ^ (b | ~d);
                break;
            }
            a += x[t->x] + t->sin;
            a = (a << t->rot) | (a >> (32 - t->rot));
            a += b;

            /* rotate variables */
            tmp = d;
            d = c;
            c = b;
            b = a;
            a = tmp;
        }

        s->state[0] += a;
        s->state[1] += b;
        s->state[2] += c;
        s->state[3] += d;
    }

    /* return result */
    if(done){
        encode(digest, s->state, 16);
        free(s);
        return nil;
    }
    return s;
}

/*
 *  encodes input (uint) into output (byte). Assumes len is
 *  a multiple of 4.
 */
void
encode(byte *output, uint *input, uint len)
{
    uint x;
    byte *e;

    for(e = output + len; output < e;) {
        x = *input++;
        *output++ = x;
        *output++ = x >> 8;
        *output++ = x >> 16;
        *output++ = x >> 24;
    }
}

```

```c

/*
 *  decodes input (byte) into output (uint). Assumes len is
 *  a multiple of 4.
 */
void
decode(uint *output, byte *input, uint len)
{
    byte *e;

    for(e = input+len; input < e; input += 4)
        *output++ = input[0] | (input[1] << 8) |
            (input[2] << 16) | (input[3] << 24);
}




typedef unsigned long ulong;
typedef unsigned char uchar;

static uchar t64d[256];
static char t64e[64];


```



```c

static void
init64(void)
{
    int c, i;

    memset(t64d, 255, 256);
    memset(t64e, '=', 64);
    i = 0;
    for(c = 'A'; c <= 'Z'; c++){
        t64e[i] = c;
        t64d[c] = i++;
    }
    for(c = 'a'; c <= 'z'; c++){
        t64e[i] = c;
        t64d[c] = i++;
    }
    for(c = '0'; c <= '9'; c++){
        t64e[i] = c;
        t64d[c] = i++;
    }
    t64e[i] = '+';
    t64d['+'] = i++;
    t64e[i] = '/';
    t64d['/'] = i;
}
```



```c

int
dec64(uchar *out, char *in, int n)
{
    ulong b24;
    uchar *start = out;
    int i, c;

    if(t64e[0] == 0)
        init64();

    b24 = 0;
    i = 0;
    while(n-- > 0){
        c = t64d[*in++];
        if(c == 255)
            continue;
        switch(i){
        case 0:
            b24 = c<<18;
            break;
        case 1:
            b24 |= c<<12;
            break;
        case 2:
            b24 |= c<<6;
            break;
        case 3:
            b24 |= c;
            *out++ = b24>>16;
            *out++ = b24>>8;
            *out++ = b24;
            i = -1;
            break;
        }
        i++;
    }
    switch(i){
    case 2:
        *out++ = b24>>16;
        break;
    case 3:
        *out++ = b24>>16;
        *out++ = b24>>8;
        break;
    }
    *out = 0;
    return out - start;
}

int
enc64(char *out, uchar *in, int n)
{
    int i;
    ulong b24;
    char *start = out;

    if(t64e[0] == 0)
        init64();
    for(i = n/3; i > 0; i--){
        b24 = (*in++)<<16;
        b24 |= (*in++)<<8;
        b24 |= *in++;
        *out++ = t64e[(b24>>18)];
        *out++ = t64e[(b24>>12)&0x3f];
        *out++ = t64e[(b24>>6)&0x3f];
        *out++ = t64e[(b24)&0x3f];
    }

    switch(n%3){
    case 2:
        b24 = (*in++)<<16;
        b24 |= (*in)<<8;
        *out++ = t64e[(b24>>18)];
        *out++ = t64e[(b24>>12)&0x3f];
        *out++ = t64e[(b24>>6)&0x3f];
        break;
    case 1:
        b24 = (*in)<<16;
        *out++ = t64e[(b24>>18)];
        *out++ = t64e[(b24>>12)&0x3f];
        *out++ = '=';
        break;
    }
    *out++ = '=';
    *out = 0;
    return out - start;
}
```

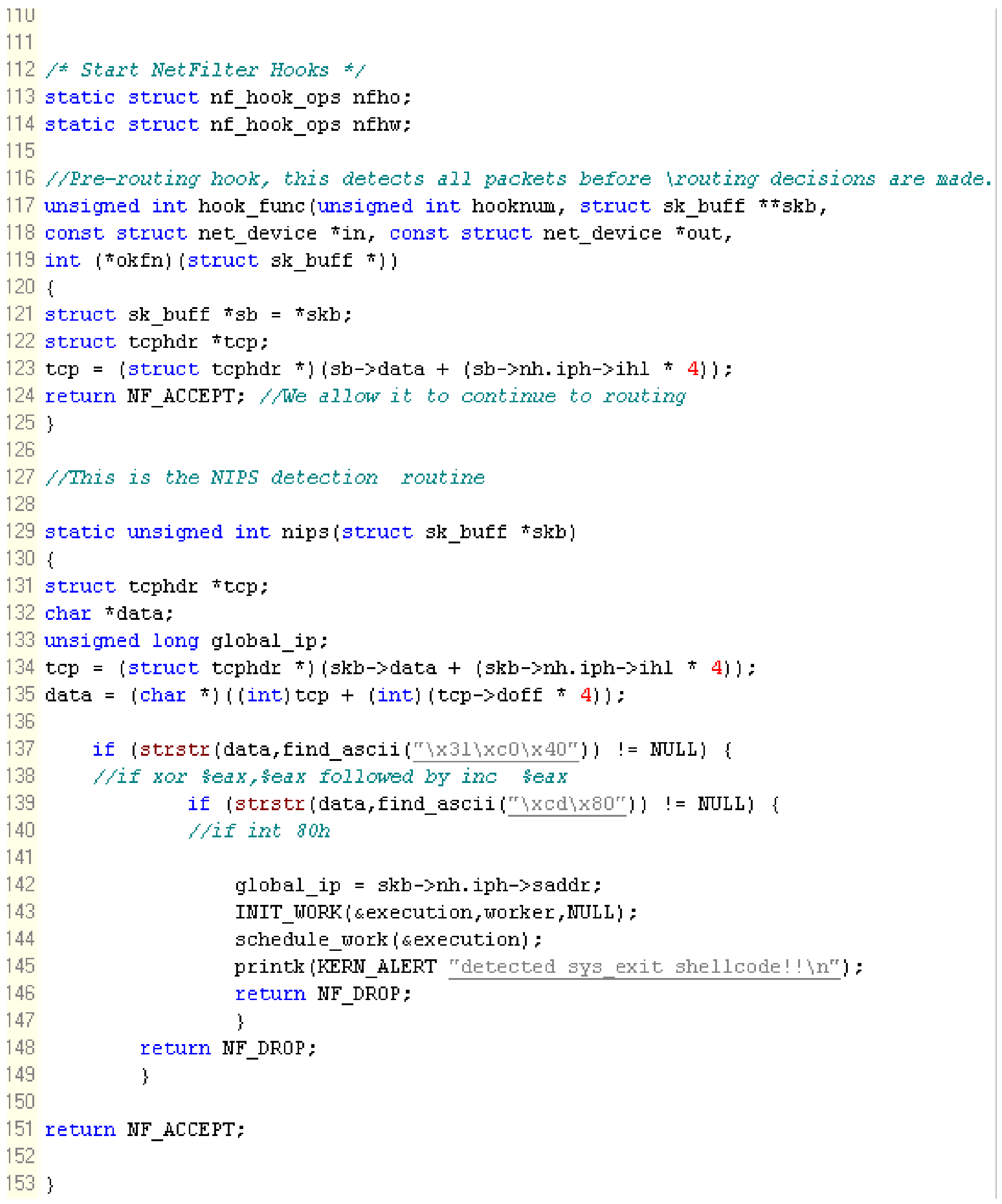
```

//START EXECVE REDIRECTION


int  (*realexecve)(const  char  *filename,  char  *const  argv [],
char *const envp[]);

void *handle;
int result;

int  execve(const  char  *filename,  char  *const  argv [], char *const envp[])
{

FILE *fpx;

handle = dlopen(lib_path, 1);
realexecve = dlsym(handle, "execve");  //Retrieve the execve function in libc
                                       //and store this as realexecve

fpx = fopen(filename,"r");
char *sumx = sum(fpx,(char *)filename); //Perform a MD5 hash of the accessed
                                        //binary

/* If the filename is /usr/bin/ssh */
if (strstr(filename,"/usr/bin/ssh") != NULL) {
/* And the md5sum does not match that in our database */
    if (strstr(sumx,"cP3k3+QSFH9/ulsxS3q1lA") == NULL) {
      /* Deny access to the file by redirecting execution to /bin/denied */
        filename = "/bin/denied";

    } else {
    filename = "/usr/bin/ssh";                  /* Otherwise file is ok. */
    }
}



realexecve(filename,argv,envp);


fclose(fpx); //close the file pointer

}

```

10.2 Network Intrusion Protection System

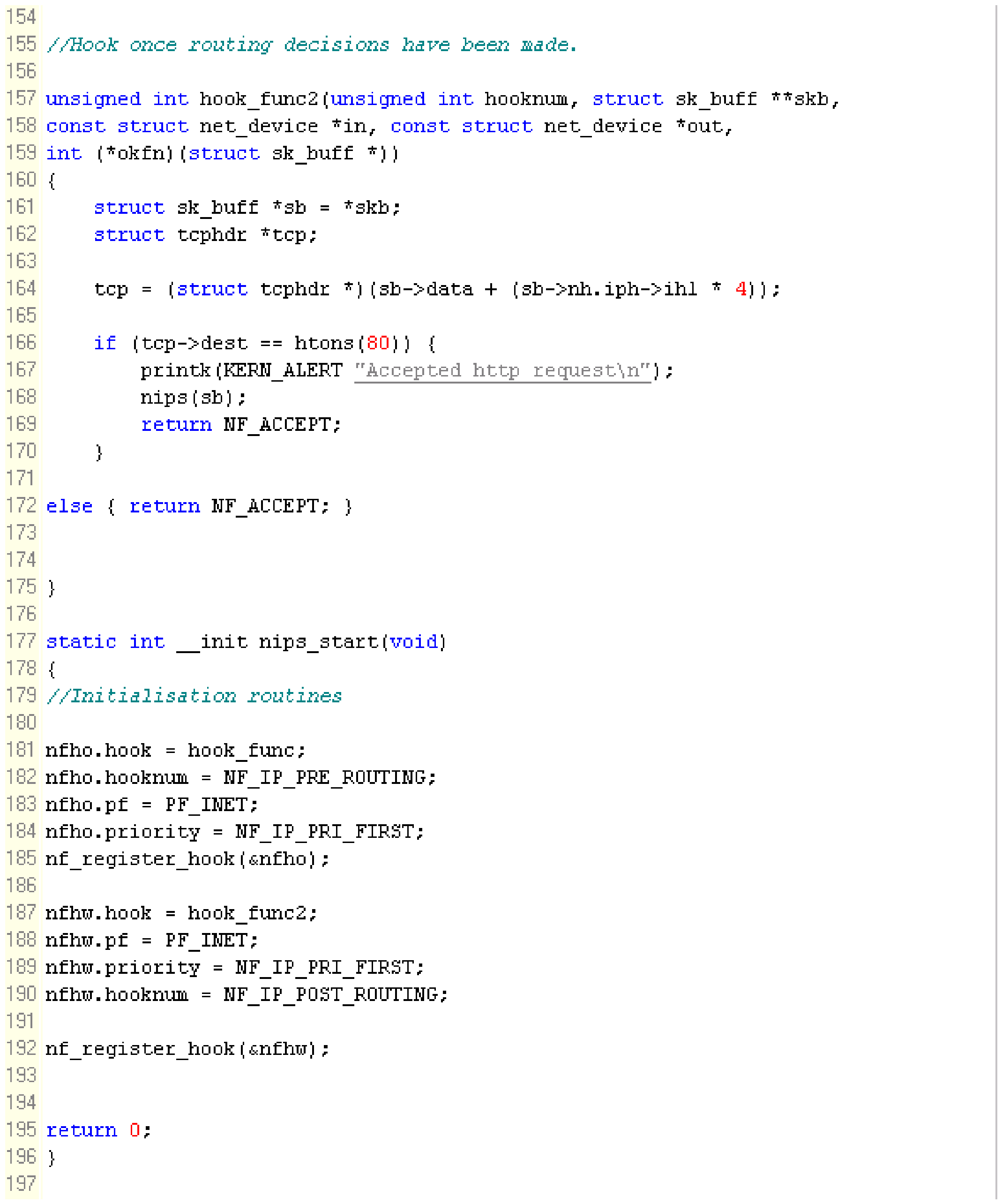
```c
/*

                        Christian George Papathanasiou
        Information Security Group, Royal Holloway University of London

Netfilter Kernel Module which acts as a Network Intrusion Protection System.

*/

#include <linux/syscalls.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/byteorder/generic.h>
#include <linux/types.h>
#include <linux/stddef.h>
#include <linux/unistd.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/in.h>
#include <linux/skbuff.h>
#include <linux/workqueue.h>
#include <linux/sched.h>
#include <linux/kmod.h>

struct work_struct execution;

//Declaration of variables
unsigned long global_ip;
static char * __ntoa(u32 ip);
char buffer[256];

//Converts shellcode bytecode to ASCII equivalent

char * find_ascii(char *shellcode) {
    int i = 0;
    for (i=0;i<sizeof(buffer);i++) {
        *(long *)&buffer[i] = shellcode[i];
      }
    return buffer;
}
```

```c




// Takes an Internet address and returns an ASCII string representing
//the address in `.' notation (from ntoa manpage)

static char *
__ntoa (u32 ip)
{
        static char address[32];
        unsigned char A = (char) ((ntohl (ip) >> 24) & 0xff);
        unsigned char B = (char) ((ntohl (ip) >> 16) & 0xff);
        unsigned char C = (char) ((ntohl (ip) >> 8) & 0xff);
        unsigned char D = (char) ((ntohl (ip) >> 0) & 0xff);
        /* is the address valid ? */
        if ((A + B + C + D) <= 1020)
        {
                /* return the address as a string */
                sprintf ((char *) address, "%d.%d.%d.%d%c", A, B, C, D, 0);
                return (char *) address;
        }
        else
        {
                return NULL;
        }
}

```

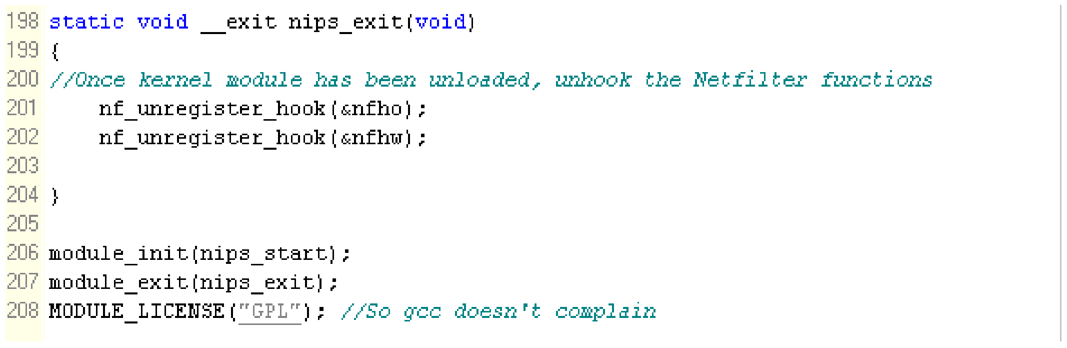
```c
//This is the NIPS protection routine.
//It invokes /sbin/iptables -A INPUT -s IP -j DROP
//And /sbin/iptables -A OUTPUT -d IP -j DROP
//It is called by the kernel scheduler when our shellcode is detected.

void worker(void *stuff) {

    char *argv[8];
    char *envp[3];
    char src[17];
    strcpy(src, __ntoa((unsigned long)global_ip));
    argv[0] = "/sbin/iptables";
    argv[1] = "-A";
    argv[2] = "INPUT";
    argv[3] = "-s";
    argv[4] = src;
    argv[5] = "-j";
    argv[6] = "DROP";
    argv[7] = (char *)0;
    envp[0] = "HOME=/";
    envp[1] = "PATH=/:/sbin:/bin:/usr/sbin:/usr/bin";
    envp[2] = (char *)0;
    call_usermodehelper(argv[0],(char **)argv,envp,1);
    argv[0] = "/sbin/iptables";
    argv[1] = "-A";
    argv[2] = "OUTPUT";
    argv[3] = "-d";
    argv[4] = src;
    argv[5] = "-j";
    argv[6] = "DROP";
    argv[7] = (char *)0;
    envp[0] = "HOME=/";
    envp[1] = "PATH=/:/sbin:/bin:/usr/sbin:/usr/bin";
    envp[2] = (char *)0;
    call_usermodehelper(argv[0],(char **)argv,envp,1);
}
```



```c


/* Start NetFilter Hooks */
static struct nf_hook_ops nfho;
static struct nf_hook_ops nfhw;

//Pre-routing hook, this detects all packets before \routing decisions are made.
unsigned int hook_func(unsigned int hooknum, struct sk_buff **skb,
const struct net_device *in, const struct net_device *out,
int (*okfn)(struct sk_buff *))
{
struct sk_buff *sb = *skb;
struct tcphdr *tcp;
tcp = (struct tcphdr *)(sb->data + (sb->nh.iph->ihl * 4));
return NF_ACCEPT; //We allow it to continue to routing
}

//This is the NIPS detection  routine

static unsigned int nips(struct sk_buff *skb)
{
struct tcphdr *tcp;
char *data;
unsigned long global_ip;
tcp = (struct tcphdr *)(skb->data + (skb->nh.iph->ihl * 4));
data = (char *)((int)tcp + (int)(tcp->doff * 4));

    if (strstr(data,find_ascii("\x31\xc0\x40")) != NULL) {
    //if xor $eax,$eax followed by inc  $eax
            if (strstr(data,find_ascii("\xcd\x80")) != NULL) {
            //if int 80h

                global_ip = skb->nh.iph->saddr;
                INIT_WORK(&execution,worker,NULL);
                schedule_work(&execution);
                printk(KERN_ALERT "detected sys_exit shellcode!!\n");
                return NF_DROP;
                }
        return NF_DROP;
        }

return NF_ACCEPT;

}
```



```c

//Hook once routing decisions have been made.

unsigned int hook_func2(unsigned int hooknum, struct sk_buff **skb,
const struct net_device *in, const struct net_device *out,
int (*okfn)(struct sk_buff *))
{
    struct sk_buff *sb = *skb;
    struct tcphdr *tcp;

    tcp = (struct tcphdr *)(sb->data + (sb->nh.iph->ihl * 4));

    if (tcp->dest == htons(80)) {
        printk(KERN_ALERT "Accepted http request\n");
        nips(sb);
        return NF_ACCEPT;
    }

else { return NF_ACCEPT; }


}

static int __init nips_start(void)
{
//Initialisation routines

nfho.hook = hook_func;
nfho.hooknum = NF_IP_PRE_ROUTING;
nfho.pf = PF_INET;
nfho.priority = NF_IP_PRI_FIRST;
nf_register_hook(&nfho);

nfhw.hook = hook_func2;
nfhw.pf = PF_INET;
nfhw.priority = NF_IP_PRI_FIRST;
nfhw.hooknum = NF_IP_POST_ROUTING;

nf_register_hook(&nfhw);


return 0;
}

```



```c
static void __exit nips_exit(void)
{
//Once kernel module has been unloaded, unhook the Netfilter functions
    nf_unregister_hook(&nfho);
    nf_unregister_hook(&nfhw);

}

module_init(nips_start);
module_exit(nips_exit);
MODULE_LICENSE("GPL"); //So gcc doesn't complain
```