Securing Distributed Computing Against the Hostile Host

John H. Hine **Paul Dagger**

School of Mathematics and Computer Science
Victoria University of Wellington,
PO Box 600, Wellington, New Zealand,
Email: {John.Hine, Paul.Dagger}@vuw.ac.nz

Abstract

Distributed computing is evolving into a collection of different paradigms that involve multiple organisations. These include mobile agents, e-Science applications such as SETI@Home or Folding@Home, and grid computing. Security is a major concern in a multi-organizational setting. To date attention has focused on authentication of participants and authorization to use resources. Protection of hosts and processes executing on those hosts have been left to the local operating system security.

In this paper we consider the security of a visiting computation with respect to a possibly hostile host. Any part of the data used by a computation, the results of the computation or the code itself may represent valuable intellectual property to its owner. The correctness of the computation may be essential to some larger, critical process.

The paper presents a methodology based on anchors of trust that allowed us to study the security dependencies within a Unix like operating system. We have identified a small set of vulnerabilities that could be exploited to create a hostile host capable of attacking a visiting computation. We show that minor extensions to a processor's microcode can be used to remove these vulnerabilities. While we can never completely remove the threat of a hostile host the proposed extensions significantly increase the work required to corrupt a host.

1 Introduction

Today's computer systems spend vast quantities of their operational time idle. Modern processors are seldom utilised anywhere near to their full capacity. The number of wasted processor cycles are even greater at night when the only real activity is scheduled routine maintenance such as backup and cleanup tasks. The growth of the Internet as a common interconnecting medium has spurred the development of new paradigms such as grid computing (Foster & Kesselman 1998) and the distribution of computational tasks around the globe into schools and homes as represented by SETI@Home (Korpela, Werthimer, Anderson, Cobb & Lebofsky 2001) and Folding@Home (Shirts & Pande 2000).

A common attribute of these new paradigms is the use of one organisation's computing resources by a different organisation's programs and processes. This introduces security risks for both the hosting system and the visiting computation. The protection of the host and of other processes that may be executing on that host is well understood and catered for by local operating system security. In contrast, the question of protecting the visiting computation from a potentially hostile host is not well understood and is the theme of this paper.

The host of a computation presents various threats. A host may simply seek to copy data without disturbing the computation in any way, as a spy might photograph a document. Alternatively the host may attempt to alter the computation causing incorrect results to be produced, but without the process detecting that the computation has been altered. Finally there is always the threat of a denial of service attack.

This paper examines the requirements for a "SafeGRID environment". This is an environment in which a user can be expected to trust the execution of a program on brokered resources belonging to an organisation over which the user has no direct control. We begin by presenting some background to the development of distributed computing and the sharing of resources. We present a generic context allowing us to define essential components of a distributed computation. Section 3 then defines the problem of the *Hostile Host*. In section 4 we discuss the essential features of trust and a model for establishing the presence of trust. We then use this model in section 5 to analyse the hostile host and identify key points where our trust in a host system is severely tested. Based on that analysis section 6 presents some straightforward microcode extensions that can be used to significantly increase the confidence that a visiting process could have in its host. The paper concludes with a summary of our work and suggestions for the way forward.

2 Background

Utilising spare resources has been studied for many years. The development of the first desktop workstations saw a flurry of activity (Wang & Morris 1985, Nichols 1987, Litzkow, Livny & Mutka 1988, Bond & Hine 1991). At this time personal Unix workstations were relatively expensive and the goal was to utilise their resources in a way that did not interfere with their owner. All sharing was assumed to be within a single organisation and security was generally ignored. Indeed most systems of that era assumed a singled shared file system was available.

More recently the stupendous capacity of idle desktop workstations has become an irresistible attraction for the apparently insatiable computational requirements of modern e-Science. As paradigms such as grid computing follow the same development curve as the web (Douglis & Foster 2003), evolving from the scientific research community to the general commer-

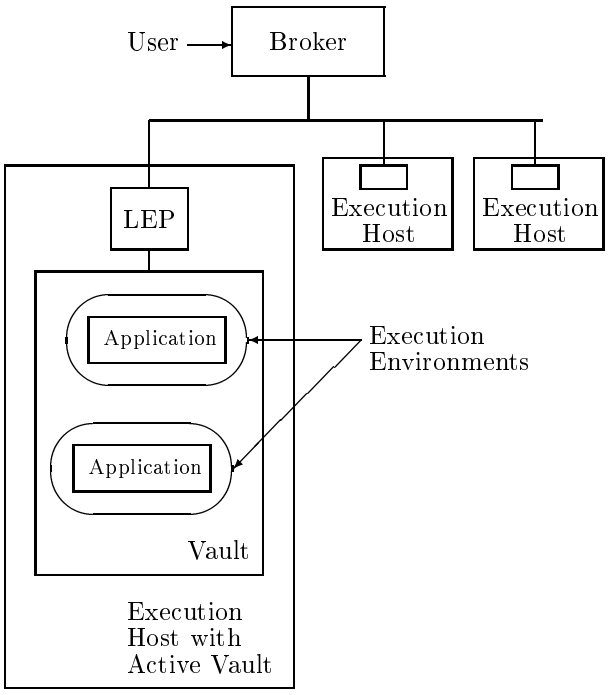

Figure 1: The Brokered Execution Environment

cial sector, we perceive the development of a market for spare computational resources. Owners of spare resources will be able to tap a new revenue source and obtain better value for money from their investment. Organisations providing a brokered distributed execution environment will be able to make profitable use of their computer systems twenty-four hours a day, typically the core business of the organisation during the day, and the brokered execution throughout the night.

Distributed execution environments that offer a means for networked hosts to be sent an application and a package of input data, process the information, and return the results already exist. The Globus computing environment (Foster, Kesselman, Nick & Tuecke 2002) has been gaining popularity and is now being marketed to organisations as a technique to leverage existing computing resources, offering savings in both time and money. Customers with applications for which they have inadequate resources are able to submit those applications for execution using a shared pool of brokered resources on an as needed basis. These execution environments either expect there to be a degree of previously established trust between the parties, or are executing applications with minimal security requirements.

Research projects such as XenoServers (Reed, Pratt, Menage, Early & Stratford 1999) and NOMAD (Bubendorfer 2001) have anticipated the development of a commercial market for computational resources. We present our work within a generic model of commercial brokering of resources shown in figure 1. The components of our model are:

User An organisation or individual with a computational requirement.

Broker An intermediary that manages the market between the user and organisations wishing to sell computational resources.

Execution host A system whose computational resources have been made available to the broker for the purpose of providing those resources to a third party.

LEP The Local End Point is a middleware program representing the broker on an execution host. The LEP is responsible for the remote loading of applications and the establishment of virtual networks between the broker and one or more execution hosts.

In an environment such as that described above it is necessary to be concerned with ensuring that application execution occurs in a trusted environment, free from the risk of compromise.

The need for a computation to be able to take place in a remote environment under the control of an unknown organisation has motivated this assessment of the requirements for the provision of a Safe-GRID environment.

3 Understanding The Hostile Host

Distributed and mobile computations are exposed to a number of risks as they execute, many are shared with traditional applications. As an example, a mobile agent and its data files can be altered or intercepted as they traverse public networks. However mobile agents executing on third-party execution hosts are not only at risk from attack by other processes on the host, but also from the host itself.

We define a "safe" host as one able to protect multiple executing processes from each other, and further is able to offer a certification that the host itself has been examined and validated as trustworthy. This can be achieved with reasonable ease when considering a distributed execution environment where all parts are owned by the same organisation, yet becomes far more complex when the mobile computation, broker, and execution hosts are all owned by distinct third-parties.

An *Hostile Host* is "unsafe". Its hardware, operating system, and utilities are in a unique position as a service provider to attack a hosted application. The threats from a hostile host can be largely grouped into five key categories; theft, alteration of program flow, alteration of input, alteration of output, or denial of service. Due to its role as system arbiter, in each of these categories a host can attack an agent either passively (e.g. by copying information as it is passed to and from memory locations without alteration) or actively (e.g. by directly altering information contained in an agent's memory locations).

The threats posed by the Hostile Host can be addressed through two inter-related techniques, *Detection* and *Prevention*. Detection involves either determining or assuming trust exists at the start of an agent's execution, and then monitoring the environment throughout execution, seeking to detect when an event occurs that would indicate a breach of the assumed trust. Examples of such events would include a copy being made of a memory page. Prevention assumes that the Execution Host is hostile at the start of execution, and will remain so throughout. As such, the hosted computation must utilise some means of preventing its data and instructions from being compromised. Examples of such mechanisms include encryption, virtual machines, monitoring for certain environment states, etc. A recurring theme throughout this work has been the objective to prevent compromise where possible, and where not, detect when a compromise occurs.

Our attempt to provide a Safe-GRID environment acknowledges that it is not possible, utilising off-the-shelf technologies, to completely solve the problem of the Hostile Host. Neither are we are trying to verify the correctness of the components of each Execution Host. Instead we seek to significantly raise the bar that a would be attacker must overcome to use

an Execution Host to mount an attack on a visiting computation. We are particularly concerned with attackers who have complete access to the Execution Host.

4 Trust

Fritz Hohl (Hohl 1998) identifies the problem of the Hostile Host as both new and unique to the field of mobile computation. The threat to a program from an Execution Host only exists if the program cannot trust the host in advance. This situation will always be true in a brokered environment because the Execution Hosts are owned by third-parties, and the program can not always guarantee which Execution Host it will be executing on.

It is also possible that the set of Execution Hosts will change over time as organisations commission new servers, retire older ones, and either join or leave the brokered system. By default mobile programs should consider their execution to be occurring on a Hostile Host. It is therefore up to the Safe-GRID system to establish trust in this trustless environment.

Three key methodologies exist with regard to the establishment of trust. The first of these, and the one more often than not employed in practice, is that described by Hohl as an *Organisational Solution*. Examples of this methodology include contracts, service level agreements, and laws in conjunction with techniques such as auditing. In general, an organisational solution relies on detection to be an effective deterrent. There is nothing preventing an entity from breaching an agreement, but should this breach be detected the agreement can be employed to invoke the prescribed punishment. Under this methodology, applications are only permitted to execute on hosts that have previously been identified as trustworthy by subscribing to the agreement.

The second methodology depends on *reputation* and assumes that all execution hosts are trustworthy until proven otherwise through appeals to a central authority. A real world example of this methodology can be seen in many of the spam 'black-hole' services. Customers of the service report potential distributors of spam to the central authority, which then decides whether to publicly advise people that the mail-server sends spam. Until a system is reported to the authority, all parties are considered equal and trustworthy. Reputation has the distinct disadvantage that by the time a host can be declared untrustworthy, the hosted computation and any information it contains have already been compromised. There also exists the problem of reliable detection when such a compromise occurs. With no other means of protection, an untrustworthy execution host could continue to be assigned applications.

The final approach involves establishing the validity of a host before execution begins, and maintaining the trustworthiness throughout the agent's execution. Should at any time it be detected that the host has become untrustworthy, the environment should be notified in order to allow the appropriate action to be taken (eg - halt execution, notify administrators, etc). Under this model, a mobile computation no longer needs to be attacked before a host can be declared untrustworthy. In order to succeed, aspects of both automated technological and organisational methodologies need to be employed. This has been the primary approach taken by the development of the Safe-GRID environment.

An Execution Host is trusted if it affords a hosted computation both protection from, and reliable detection of, threats to its execution. In order to better analyse the exact requirements, the *Root Trust Model* as shown in figure 2 has been developed. This model establishes the desired outcomes of any execution, namely correct results, analyses the threats to these results, and then establishes the required mechanisms to address them.

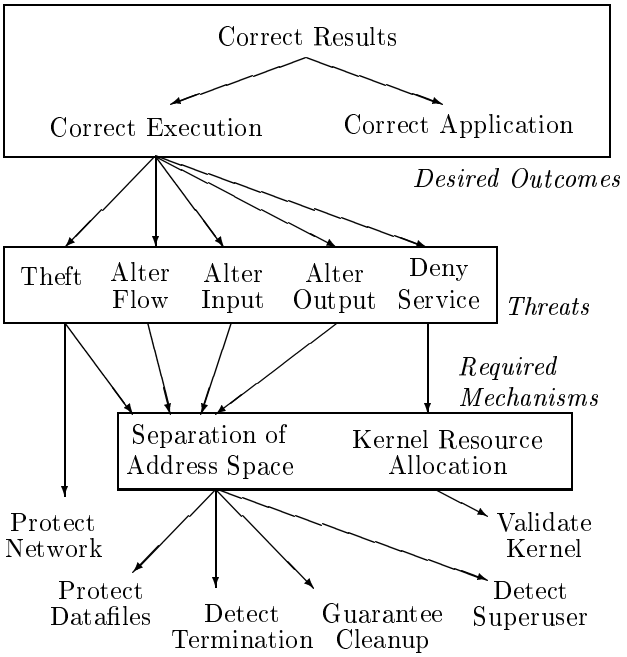

Figure 2: The Root Trust Model

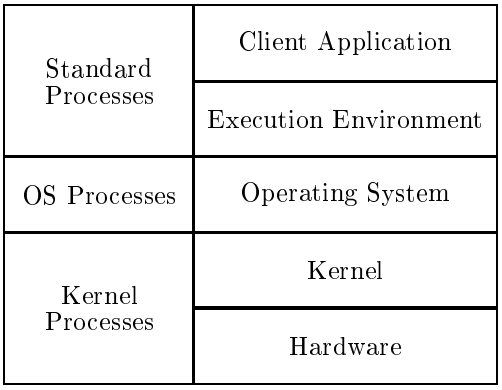

Figure 3: The Layer Model

The root trust model identifies two primary requirements that must be satisfied in order to protect a mobile computation against a Hostile Host; separation of application space, and kernel resource allocation. These two core requirements each rely on a number of sub-problems.

To identify mechanisms to solve these problems, and to clearly define the scope of each of these problems, work on Safe-GRID has broken down the establishment of trust within an execution host into a number of layers, each building and relying on those beneath it. This creates the Layer Model shown in figure 3. To ensure trust within the execution host as a whole, trust must be established within each layer in order.

The Hardware layer is concerned with the base hardware that makes up the computer system. It is assumed that this hardware will be standard off-the-shelf equipment supplied from a hardware vendor. The Kernel layer is concerned with the operating system kernel. It is assumed that a UNIX-like monolithic kernel architecture is used on the Execution Host. The OS Environment layer is made up of the various libraries, system binaries, and user appli-

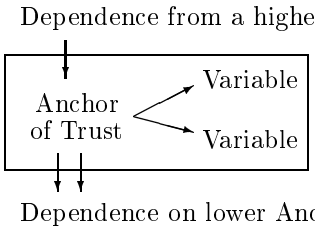

Figure 4: Relationship Between Variables and Anchors

cations that make up a UNIX-like operating system. The Execution Environment layer comprises the components specific to the distributed execution environment. From this layer up, all components are standard operating system processes and execute in parallel with other applications on the Execution Host. It is the responsibility of components in the Execution Environment layer to provide the necessary support for a remote Client Application to be submitted for execution, carry out its execution, and collect any results from this execution. Finally, the Client Application layer is concerned solely with the user's application itself. This layer is a capstone to the model. It is considered highly unlikely that a user's application would attack itself, assuming no internal sabotage or conspiracy exists within the user's organisation, doing so offers no identifiable benefit.

Anchors of Trust act as the fundamental mechanism through which trust in a particular layer is established. An Anchor of Trust is the combination of a group of variables whose validity and correctness can be proven conclusively to authenticate each variable. This proof involves both independent variables, and previously established Anchors of Trust. If valid, an anchor is able to either completely or in combination with other anchors provide the required trust in a layer. A single layer may contain multiple anchors. After the initial anchor, the proof of validity of subsequent ones may rely on previous anchors, therefore if an anchor located in the hardware layer is proven to either be initially invalid, or is detected to have become invalid, the dependent anchors in all layers above it also immediately become invalid. This relationship between variables and anchors, and multiple anchors is shown in figure 4.

This interdependence between Anchors of Trust creates the notion of a *Chain of Trust* running from the hardware layer all the way to the Client Application. As with any physical chain, the Chain of Trust is only as strong as its weakest link. The Safe-GRID research has sought to identify mechanisms that establish individual Anchors of Trust, and then determine where weaknesses exist creating exposure to threats. Where possible, mechanisms were then put in place to strengthen links in the chain. Through this, it has been determined that it is not possible to reliably establish trust within a hostile host. This can be attributed directly to the inability to conclusively prove the validity of the operating system kernel.

5 Analysing the Hostile Host

The Anchors of Trust identified as part of the Safe-GRID research can be largely divided into three categories. Those contained in the user-space, those in the kernel-space, and the hardware anchors providing support to one or both of these categories. Examples of Anchors of Trust in the user-space include the validation of Safe-GRID components, validation of user-space operating system utilities, and the protection of the computation's data files and binaries. Almost all of these are dependent on anchors in the operating system kernel such as file system security and the integrity of the system's memory management. In turn these Anchors of Trust rely on the validity of the operating system kernel and its associated boot loader. Further supporting Anchors of Trust are found in the hardware layer of the Layer Model (figure 3), and include guarantees that the Execution Host hardware will not intercept or alter memory contents, and that the processor and disk subsystems will operate in a predictable and trustworthy manner.

Due to the complexity involved in compromising the Anchors of Trust contained in the hardware layer, it has been assumed that the Execution Host's microcode and hardware subsystems are trustworthy and contain reliable anchors. Consequently the safety of our application level computation can be seen to rest squarely on our ability to demonstrate that the operating system kernel has not been tampered with. It is important to note that we are not reliant on the correctness of the operating system. We acknowledge that operating systems have bugs and are likely to continue to do so for some time. We are concerned that the operating system has not been tampered with in a manner that makes it hostile to our computation.

We are able to validate either source or executable code through the use of a digital signature. But how do we apply this to an operating system kernel? We cannot expect the kernel, or any agent of the kernel, to carry out the validation. This is impossible because of the dual problems of *Circular Validation*, and *Perfect Knowledge*. Circular Validation is the basic premise that if you ask a component to validate itself, the answer provided can not be trusted. The real world equivalent to this is the well known paradox, "everything I say is a lie". Due to Circular Validation, validation of the kernel must be carried out by a component external to and unaffected by the kernel. In a monolithic architecture, once booted, the kernel has complete control over all aspects of system operation. This therefore requires validation to occur before the kernel is started at system boot.

A possible solution to the problem of generating a signature of the kernel external to the kernel, presents itself through a number of observations regarding the functioning of a monolithic kernel operating system:

1. Once a kernel has booted, it has control of the system resources. Therefore any attempt at external kernel validation after kernel boot will suffer from the problem of Circular Validation.

2. Most UNIX style operating systems make use of a boot loader to start the kernel. This boot loader has control of all hardware resources while running and is started via the hardware and the Master Boot Record (MBR)

3. The boot loader is a small and inherently simple program that can be easily replaced with a substitute.

Due to its isolation from the kernel, the boot loader can be used to carry out the required external validation without falling into the trap of Circular Validation.

The simplest form of the kernel validation algorithm that makes use of the boot loader is therefore:

1. The hardware initialises using the Master Boot Record.

2. The secure boot loader is started.

3. The boot loader computes the MD5 checksum of the kernel and writes it to a specific disk location.
4. The boot loader boots the kernel.
5. At sometime in the future, the MD5 checksum is loaded from disk, transmitted to the broker and compared against a known value.

Until proven otherwise, the entire Execution Host must be considered untrustworthy. The decision surrounding the validity of the kernel's signature must be carried out independently of all parts of the Execution Host, i.e. it must be transmitted to the broker which maintains valid signatures of Execution Hosts under its management. This cannot happen until the operating system has begun execution. The kernel cannot be allowed to tamper with the signature between its calculation and transmission to the broker. This requires a more complex algorithm than the one given above.

In order to carry out this validation, the operation of the secure boot loader is critical. Certain requirements exist for the establishment of kernel validity:

1. Only the secure boot loader may generate the values used to prove validity
2. The output must be immutable for a defined period of time allowing transmission to the broker.
3. The output values must be revoked automatically when the anchors of trust they are based on are removed
4. An anchor of trust must exist within the boot loader

It is clear that with the MD5 checksum stored in plain text on the host system, it would be a trivial matter for a malicious kernel that knows the correct MD5 checksum, to replace the computed value with one that indicates to the broker that a valid kernel is running. To counter this risk, it is necessary to encrypt and sign the MD5 checksum in such a way that no parties other than the boot loader and broker are able to decrypt and access the value. This would also stop the kernel from substituting a false checksum value.

While protection is now provided for the initial kernel validation, once a successful boot and validation has occurred, the owners of the host system could then make a copy of the encrypted checksum and make use of it as part of a replay attack. After booting an invalid kernel, the copied encrypted checksum is merely re-transmitted to falsely indicate a valid kernel was started.

To counter this, the simple encrypted MD5 checksum is replaced with a multi-value encrypted ticket that is unique for each boot. The ticket incorporates a time-value identifying the time the system was booted (t), and the result of a mathematical combination involving the time-value, a unique serial number incremented on each boot and the MD5 checksum (H). The ticket is expressed as T(t,H). In addition to the ticket being passed to the broker, the time that the system was booted must also be passed.

To encrypt the ticket in such a way that the host itself is not able to decrypt or replicate it, it is necessary that a shared secret exist between the boot loader and the external validation agent. This brings us to the problem of perfect knowledge. A Hostile Host will have full access to the data and programs stored on that host. It will have perfect knowledge of its environment. Hohl has shown that any secret deployed into a hostile environment long enough will eventually be broken. (Hohl 1998) A means of providing protection for the copy of the secret stored within the boot loader needs to be found.

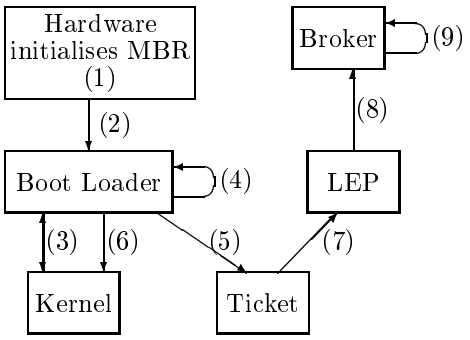

Figure 5: Kernel Validation Algorithm

Previous work done by Hohl (Hohl 1998) in the field of time limited blackboxes has identified the use of obfuscation to disguise the operation of an application deployed into an untrustworthy environment for a specific period of time. It would therefore be possible to make use of these techniques to hide a secret shared between the customised boot loader and an external authentication agent. This secret is initially provided through a trustworthy delivery mechanism (eg - 'sneaker-net') and is then replaced, when necessary, securely through the LEP to Broker communications channel.

We can extend our earlier algorithm to incorporate the use of unique tickets. The steps of the algorithm are illustrated in figure 5.

1. The hardware initialises using the Master Boot Record.
2. This starts the modified boot loader.
3. The boot loader computes the MD5 checksum of the kernel and notes the system time (t).
4. The system time, secret identification, and MD5 checksum are mathematically combined to produce H.
5. A ticket, T(t,H), is produced and encrypted before being written to disk as Eb_s[T(t,H)] where 's' identifies the secret used to encrypt the ticket.
6. The boot loader boots the kernel.
7. The LEP is executed and retrives the ticket.
8. The LEP then transmits Eb_s[T(t,H)] and t to the broker for validation.
9. The external validation agent decrypts the ticket, uses its known good MD5 checksum, s and t to recompute H and compares it to that sent from the host to determine validity.

It is now possible to reliably (for a period of time), generate a ticket that is able to validate the running kernel and can not be reproduced by the kernel itself. The threat of the trivial replay attack has also been addressed through the introduction of a time-stamp within the ticket that is mathematically integrated with the MD5 checksum. A threat, albeit a difficult one to exploit, still exists from a non trivial replay attack in which the system clock is adjusted between a valid boot and an invalid boot, allowing the ticket to be replayed. To counter this, an outstanding problem still exists in finding a reliable way of uniquely identifying each boot.

A significant weakness within the algorithm is the reliance on a time limited black box to contain the boot loader's secret. This is one of the key factors preventing reliable kernel validation. The question has to be asked as to how long the security of such a secret deployed into a hostile environment can be maintained? What may take one host a day to compromise, could take another host only a matter of minutes. Another issue with a blackbox protected secret is that considering the small size of the boot loader, there are only a finite number of distinct ways that a hidden secret could be masked. After a significant number of variations, it is expected that patterns would begin to emerge, aiding the task of compromise.

Re-examining the role of the secret in this algorithm, a series of requirements that any potential mechanism must satisfy are apparent:

1. As a means of encrypting the validation ticket, the secret ensures that a fraudulent ticket can not be reproduced
2. As a valid ticket can only be generated with the valid secret, the secret also provides a means of ensuring that the modified and trusted boot loader started the system.

It is important to note that due to the boot loader's deployment on a potentially hostile host, a would be intruder will have perfect knowledge regarding the function of the boot loader, and any algorithms and data used within any other system component. This knowledge means that it is possible for any mechanism used to prove the validity of the boot loader to be replicated by a hostile party. Once replicated, an invalid boot loader may start an invalid system kernel and disguise this by providing results from any validation mechanism that falsely indicates that no problems exist. The only defense to the problem of perfect knowledge is the amount of time that is required to take advantage of it. This is the principle behind the use of a regularly replaced component, by the time a would-be intruder has determined its operation it has been replaced.

Once generated at boot, a ticket must remain secure on an Execution Host until the first time the environment is instantiated. This critical section is illustrated in figure 6. Steps one to nine from figure 5 take place between t_0 and t_1 producing and validating a ticket based on secret s_1. At a later time a new secret, s_2, is introduced. The integrity of this secret must be maintained from its introduction until the ticket produced during the next boot operation has been validated at t_5. Once this has been done and the unique boot identifier stored within the ticket associated with a valid kernel boot, each time the environment is started the boot identifier previously identified as required merely needs to be checked to determine if the anchor of trust still exists.

Considering that it is likely that any host providing computing resources to the distributed execution environment is likely to run at least one client application every twenty-four hours, in the general case a ticket (and the secret used to generate it) would have to remain valid for up-to twenty-four hours after the boot. Based on this observation, a secret must remain secure from the time it is downloaded, until it either expires or a ticket generated by it is used to validate a particular boot. However it is conceivable that a host may not run a client application for a much greater period of time.

Due to the problem of perfect knowledge, a secret shared between the Broker and the Boot Loader is necessary to provide the anchor of trust. This secret must be protected against compromise by the execution host, a time limited blackbox can achieve this.

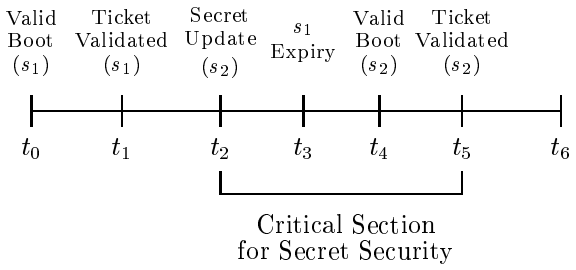

Figure 6: Ticket Security Critical Timeline

This Blackbox is not a sufficient mechanism for allowing kernel validation. Consequently the anchor of trust that is generated in this layer is significantly weaker than all others.

The Kernel layer is the key layer in terms of establishing trust and therefore protection against the Hostile Host, and unfortunately the layer where the weakest mechanism for establishing this trust exists. In order to establish anchors in higher layers, a valid kernel must be booted. This requires its own anchor in the form of a valid boot loader. This is a requirement that can not be reliably provided using off-the-shelf technologies. A number of potential enhancements to existing architectures have been developed as part of Safe-GRID which address the limitations preventing kernel validation. These enhancements are discussed in the next section.

6 Raising the Bar

The inability to reliably validate the kernel, the central arbiter of the operating system, results in all higher anchors of trust immediately becoming invalid. In order to address this risk and 'raise the bar' sufficiently, it is critical to the establishment of the required level of trust that the kernel can be validated.

The key problem preventing kernel validation is the inability for any secret used in the technique to be kept secret. This results in a hostile kernel being able to forge legitimate validation components. An ideal solution would be one which required no secret, yet produced validation components that can not be replicated. Further, a trustworthy means of generating a unique boot identifier still must be provided.

It has been assumed that without making radical changes to the hardware, it would not be possible for a would-be intruder to passively copy the contents of either the memory or processor in hardware, and similarly it would be very difficult for instructions and memory contents to be altered in such a way that they still made sense to a client application, the function of which is not known at the time of compromise.

All operating systems must be initially started by the system hardware and it is a reasonable assumption that it is a difficult task to alter hardware components at the microcode level. Using these facts, it is possible that the microcode could be employed to obtain the checksum of the kernel being started. Checksum algorithms such as MD5 are well defined and may be easily implemented in microcode. The output of this validation can then be stored in a register which is read-only to any entity other than the validation hardware. Provided this register can be read reliably by applications, and without the assistance of the kernel, Safe-GRID components can rely on the publicly accessible, secure checksum to validate the kernel.

In order to compromise a validation technique based on the hardware validation register, either ad-

justments would need to be made to the microcode producing the digital signature, or interception of requests for information from the register and alteration of the results. When considering the hardware layer, it was argued that the required level of trust could be obtained by the assumption that significant and highly technical work would be required to alter the operation of the hardware to cause a compromise. This combined with the ability for assembly language code to access the processor directly, suggests that it is unlikely such a compromise could occur.

To allow the trustworthy identification of an individual boot, a means needs to be found to provide unique identification via an immutable mechanism, the value of which would replace the time-stamp in the ticket. As it is assumed that the operating kernel can only be replaced through a reboot, a valid ticket should be immediately rendered invalid by the replacement process. In order to find a solution to this problem, it first needs to be assumed that the hardware layer contains a correct anchor of trust and can therefore be relied on.

By introducing a single read-only 32 bit register into the hardware platform, this can be achieved. During the initial initialisation of the host system, the contents of this register cycle randomly. At the instant that the boot loader is started by the hardware, the current value of the register is set creating a random 32 bit number. This provides a value to be stored in the ticket that is unlikely to re-occur. This hardware based *Boot Identifier* significantly reduces the threat from a non-trivial ticket replay attack.

The addition of these two read-only registers removes the need to transmit a ticket encrypted with a secret shared by the boot loader and the broker. A hosted Execution Environment can be created by a LEP which has read both the kernel's checksum and the random identifier. These can be transmitted to the broker to determine if the system has rebooted since the last environment was established and if the checksum of the kernel matches the value stored by the broker. The kernel cannot modify these values and therefore a modified kernel is not able to hide its identity.

These changes would also prove beneficial to users of the system by allowing for clear identification of a particular operating system boot, and the exact variant of a kernel that was started. This can be used by system administrators to track potential problems to individual boots, the exact kernel configuration that was in operation, and precisely when a reboot occurred.

At initial inspection, it would appear that with these architectural changes, the Safe-GRID system is able to address the problem of the Hostile Host. This is not necessarily the case. We have aimed through this research to 'raise the bar' with respect to the hurdles a would-be attacker must overcome. As such, it can not be guaranteed that a mobile agent is perfectly secure, in fact perfect security is an established hyperbole.

The solution presented in this paper also relies heavily on the assumed trustworthy nature of microcode. Whilst a reasonable assumption, it leaves an identified weakness in the chain of trust.

7 Conclusion

The Safe-GRID research set out to determine the validity of Hohl's description of the problem of the hostile host as "insoluble" (Hohl 1998). It has been shown that this is true with regards to current off-the-shelf technology, primarily due to the problem of a hostile agent's perfect knowledge regarding the execution host. Safe-GRID has sought to 'raise the bar' sufficiently to ensure that the effort required to compromise the environment is beyond all but the most determined and well resourced attackers.

This paper has presented a model which allows trustworthy execution to occur based around the establishment and maintenance of 'anchors of trust'. Investigation of this model as part of the Safe-GRID research has highlighted a key factor hindering the establishment of a trustworthy execution environment. This is the difficulty in validating the kernel due to the problem of perfect knowledge regarding both the Execution Host, and given sufficient time and resources, the Execution Environment.

We have then presented two relatively simple microcode based architectural changes, which if implemented would significantly strengthen techniques used to validate the kernel, and therefore establish a chain of trust. These proposed changes also offer beneficial enhancements for other users of the Execution Host.

Further research is still required to investigate both the feasibility and reliability of relying on microcode in this manner, however if successful would overcome the last major obstacle to addressing the problem of the hostile host, and establishing trust in a trustless environment.

References

Bond, A. & Hine, J. H. (1991), Drums: A distributed performance information service, *in* J. Lions, ed., '14th Australian Computer Science Conference', ACS, Sydney, pp. 34.1–34.9.

Bubendorfer, K. (2001), NOMAD: Towards an Architecture for Mobility in Large Scale Distributed Systems, PhD thesis, Victoria University of Wellington.

Douglis, F. & Foster, I. (2003), 'The grid grows up', *IEEE Internet Computing* **7**(4), 24–26.

Foster, I. & Kesselman, C., eds (1998), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann.

Foster, I., Kesselman, C., Nick, J. M. & Tuecke, S. (2002), 'Grid services for distributed system integration', *IEEE Computer* **35**(6), 37–46.

Hohl, F. (1998), Time limited blackbox security: Protecting mobile agents from malicious hosts, *in* G. Vigna, ed., 'Mobile Agents and Security', Springer-Verlag, pp. 92–113.

Korpela, E., Werthimer, D., Anderson, D., Cobb, J. & Lebofsky, M. (2001), 'SETI@home: Massively distributed computing for SETI', *Computing in Science and Engineering* **3**(1), 78–83.

Litzkow, M., Livny, M. & Mutka, M. (1988), Condor - a hunter of idle workstations, *in* 'Proceedings of the 8th International Conference of Distributed Computing Systems'.

Nichols, D. (1987), Using idle workstations in a shared computing environment, *in* 'Proceedings of the Eleventh ACM Symposium on Operating Systems Principles', ACM, Austin, TX, pp. 5–12.

Reed, D., Pratt, I., Menage, P., Early, S. & Stratford, N. (1999), Xenoservers: Accounted execution of untrusted code, *in* M. Satyanarayanan, ed., 'IEEE Hot Topic In Operating Systems (HotOS) VII', IEEE Computer Society, pp. 136–141.

Shirts, M. & Pande, V. (2000), 'Screen savers of the world, unite!', *Science* **290**(5498), 1903–1904.

Wang, Y. & Morris, R. (1985), 'Load sharing in distributed systems', *IEEE Transactions on Computers* **C-34**(3), 204–17.