# Root Kits
# – An Operating Systems Viewpoint –

Winfried E. Kühnhauser

University of Ilmenau
winfried.kuehnhauser@tu-ilmenau.de

**Abstract**

Root Kits are tool boxes containing a collection of highly skilled tools for attacking computer systems. Their algorithms and databases contain professional knowledge about methods and mechanisms for completely automated attacks both over a network as well as from within a system. Root kits attack by maneuvering a system into executing a script with supervisor privileges. Once having gained full control, such scripts begin to install several software packages, including backdoors for easy future access, deception packages and modified versions of administration utilities that conceal system modifications and refuse to counterattack any future infiltration.

The security threat imposed by root kits is quite serious. A root kit attack is swift, fully automatic, and has long-lasting effects. An attack has a high success probability, and it requires only a very small amount of knowledge. Last not least, root kits are easily available in the Internet.

This paper is a survey of the works of root kits from an operating systems point of view.

**Keywords**: error exploitation, error proliferation, privilege proliferation, kernel abstractions, trusted computing base, reference monitor, security domains, mandatory and discretionary access control, secure booting, secure program execution

## 1 Introduction

Due to the large amount of knowledge contained in their algorithms and databases, the threat imposed by root kits for today's industrially used computer systems is very high. A root kit attack consists of a systematic weak point analysis and a case-specific assembly of an attack suite including case-specific concealment techniques. After a successful infiltration, several backdoors are installed in order to obtain an all-time, indiscoverable and complete control over the attacked system. Additionally, these backdoors are concealed by deception programs that substitute standard administration and maintenance programs as well as operating system components. Every step of an attack is performed completely automatic and extremely fast.

The high success probability of a root kit attack is the result of a comprehensive and automated weak point analysis. Root Kits exploit mistakes and errors that generally are hard to avoid in today's complex system software. Error causes are numerous, ranging from specification mistakes and implementation errors within operating systems, servers and utility programs up to mistakes in the configuration and administration of a system. In general, a single exploitable mistake is sufficient to open the gate, and the large number of existing mistakes known to a root kit's database makes its discovery highly probable.

For error discovery root kits command an extensive database of today's contemporary mainstream operating systems, system-level services and utilities along with their versions, errors and weaknesses. A systematic analysis based on such a database is very often successful. On the one hand many systems are

running services which they actually do not need[1] which makes the probability of hidden errors very large. On the other hand there often is a considerable gap between the time an error in some service or operating system becomes publicly known and the time when it is corrected. Few system owners can afford such rigorous actions as to close down a faulty system or service even if they are informed about the hovering threats. By exploiting mistakes and errors root kits use system weaknesses that are extremely difficult to prevent, and that may be re-introduced again and again whenever systems are updated or extended.

The actual attack then proceeds so fast that it is virtually impossible for human observers to detect it, let alone to be able to take counteractive means. The completeness of control gained over the attacked system allows a root kit to conceal itself immediately after the gate is open. It then starts installing backdoors and deception software that hides these backdoors as well as their future opening. Even in those rare cases in which some backdoors are detected, without further precautions one can never be sure to have detected them all. And even in those rare cases in which the opening of a backdoor in the course of a future attack is detected, modified versions of standard administration utilities installed by the root kit prevent any counteractions.

Last but not least, the more powerful the attacked system the more brief is also the actual attack time. Once more root kits use very fundamental qualities of the attacked system for their malicious purposes.

# 2 Attack Technique

A root kit attack consists of four completely automated steps.

The first step is the vulnerability analysis. In this step, a root kit searches for weaknesses in the operating system as well as in high-privileged server processes. To this end, a chronologically and numerically randomized scan of standard IP ports is performed by ways of harmless appearing requests. As a result, the root kit obtains information about the operating system, its version and a list of running services. On a networked system such services are quite numerous, encompassing SSH (*Secure Shell*), NTP (*Network Time Protocol*), FTP (*File Transfer Protocol*), or Web servers, to mention only a few. A root kit then looks up its vulnerability database in order to select the most convenient weakness.

The second step is the vulnerability exploitation. Based on the result of step one, the goal is to bully the vulnerable system component into executing code of the attacker. This code then serves as a bridgehead and creates an execution environment in which the root kit may execute scripts running with high privileges. In Unix systems, root kits strive for "root" privileges, which also gave them their name.

Once having adopted root privileges, the third step erases all traces of the current attack from the system's log files. Furthermore, the root kit's active and passive components are hidden by exchanging utility programs that inform system administrators about the state of the file system and running processes by preassembled fakes. In Unix systems, programs such as *ps, pstree, top* or *ls* are exchanged by modified versions containing filters that eliminate root kit processes and files from their output. More recent root kits also directly manipulate registries, the Unix */proc* directory or the operating system itself.

The last step installs one or more backdoors by which the attacker will obtain access to the system even if the originally exploited vulnerability is long since removed. Several opportunities for installing these backdoors exist. They can be hidden for example in modified versions of standard server programs and demons (such as *sshd* or *ftpd*), in authentication programs (such as *slogin*) or in their dynamically linked libraries (such as LDAP libraries used by the *login* program for checking passwords).

The existence of these backdoors is obscured by modifying the modification date of affected program files, by filter mechanisms such as mentioned in step three or by directly modifying the operating system's API implementation (e.g. for the Unix *stat* or *getdents* system calls).

[1] in a standard Sun Solaris installation twenty-nine servers are started during system initialization

The entire attack including the installation of backdoors, the covering of tracks of the current attack and the installation of the concealment mechanisms for hiding future "visits" typically lasts only a few seconds, a time in which even an attentive human observer has no chance for counteractions. Ironically, the more powerful the attacked system is the shorter the actual attack time will be. Then, after only a few seconds the attacked system is in a state which appears to be normal even for quite experienced system administrators.

# 3 Tools in a Root Kit

The tools in a root kit serve for the analysis of vulnerabilities, the concealment of attacks and the prevention of the discovery and counteraction of future attacks. They include algorithms to establish contact with the attacked system and to evaluate its response, data bases with known vulnerabilities of operating systems and services, and preassembled fakes of standard utility programs and operating system components for the replacement of their originals.

## 3.1 Tools for Vulnerability Discovery

Tools for vulnerability discovery (also called exploiters) use a data base containing three different classes of vulnerabilities:

- vulnerabilities of standard utilities and server programs; examples are utility programs such as *ssh/slogin, mount, cron, sendmail, lpr, ftp*, servers running as demon processes such as *sshd, maild, inetd, nscd, smbd*, or web browsers and servers; typically, implementation errors such as weak parameter type checks, unchecked buffer or stack overflows and race conditions are exploited by provoking stack or buffer overflows, calling functions with illegal argument types or moving program files while regular scripts are running
- vulnerabilities of the version of the current operating system caused by similar implementation flaws
- vulnerabilities of the system configuration such as default passwords or open communication ports that were left unchanged during the initial system installation and configuration.

## 3.2 Tools for Covering Attacks

The tools in this group aim at hiding all hints to the root kit attack and are threefold. Firstly, they aim at covering all traces of an ongoing attack. Secondly, they hide modifications to system, servers, libraries and utilities that have been made in the course of the attack. Thirdly, they make provisions for hiding activities that will be started when the system is accessed in the future by one of the installed backdoors. Tools in this group hide

- root kit processes running in the course of an ongoing attack
- root kit processes running in the course of using a backdoor
- active network connections
- root kit files installed during the attack
- the exchange of code files fakes containing backdoors
- the restart of servers that crashed because their configuration files have been manipulated

- the restart of servers whose code files have been backdoor'ed.

Techniques applied to this end enclose

- removal of log file entries about processes or network connections that have been started or opened in the course of the attack
- modification of all administrational utilities that inform about running processes, active network connections, and state of the network interfaces and file systems.

Moreover, because of the if low but still existing possibility of a later backdoor detection newer root kits modify dynamically linked libraries as well as dynamically loadable operating system modules to suppress information about the process system, the file system and the communication system immediately at their source.

## 3.3 Tools for Preparing Future Attacks

A successful root kit attack strives for sustainability. Even in those cases where the initially exploited vulnerability is mended there still is supposed to exist a hidden entrance to the attacked system.

To this end root kits install back doors in standard utility programs. As an example, there is a backdoor'ed version of *sshd* that contains a code sequence which carries out shell commands with root privileges if the local user name contains a secret pattern known only to the attacker. By issuing the command "*ssh -l 'secret-pattern' 'attacked-host' /bin/sh -i* " the attacker will then always be able to start an interactive root shell on the attacked system.

Furthermore the already mentioned concealment mechanisms hide future attacks. In the rare cases when and attack is nevertheless detected, filtering techniques built into administration programs render any countermeasures impossible.

Similar to techniques for concealing ongoing attacks, newer root kits bury backdoors as well as covered network channels and command filters deep into the operating system kernel. It gets hair-raising if in spite of all these concealment techniques a root kit is nevertheless detected but the operating system itself refuses to remove its components.

## 3.4 Prefabricated System Components

This tool group contains preassembled manipulated versions of standard utility programs, dynamically linked libraries and dynamically loadable operating system modules. These components are installed in the course of a successful attack and hide the ongoing attack itself as well as installed root kit files and programs, backdoors, and the future opening of backdoors.

The following examples replace standard utilities and demon processes in the Unix operating system family.

- modified versions of *ls, du, find* and *md5sum* conceal modified or added software; this concealment works also for programs with cryptographic fingerprints based on the MD5 algorithm, because the verification program is replaced, too
- active network connections are hidden by modified versions of *netstat* and *ifconfig*
- active background processes are hidden by modified versions of *ps, pstree, top* and *ksysguard*
- log file laundry is performed by versions of demon processes such as *syslogd* containing laundry code

- modified versions of demon programs such as the *inet-, ssh-, ftp-* and *slogin-* demons as well as modified versions of administration programs such as *passwd* contain backdoors for future system access
- a whole bundle of spying, sniffing and response mechanisms is built into several different programs. Programs such as *login, ssh, linsniffer* or *ftp* look out for user names and passwords, other problem-specific espionage programs directly spy out database contents
- several administrational utilities are included for preventing future counteractions against the root kit itself, among them versions of *top, ksysguard, kill, killall* or *shutdown* that refuse to terminate root kit processes or to halt the system while a root kit is active or a backdoor is in use
- last but not least, similar to viruses or worms root kits may contain proliferation mechanisms for automated large scale attacks.

# 4 Defense Strategies

Root kits attack by exploiting specification flaws and implementation errors in operating systems, servers and utilities. As a consequence, as long as such exploitable flaws and errors persist there will be no safe protection against root kits. Current state of the art in software engineering considered it seems not very wise to expect that in the near future we will be capable to build large and complex software systems without making mistakes. In order to deal with root kits we thus have to focus on a different approach.

Firewalls that control the interface between Internet and a proprietary Intranet likewise have their limits [5]. On the one hand, mobile system components such as Laptops and PDAs physically bypass physical firewalls, and wireless networks impose an additional threat. On the other hand, firewalls are useless to counter insider attacks which, as experience shows, frequently are the more serious ones.

This leads to the rather unsatisfactory conclusion that vulnerabilities exploited by root kits are inherent to today's systems, and that there is a fundamental difficulty to eliminate these weaknesses. The general approach for dealing with root kits thus must follow two roads. On the one road efforts to write correct and manageable systems must be continued. The second road on the other hand accepts the existence of errors and strives for putting limits to their exploitability.

## 4.1 Prevention

The goal prevention is to minimize the probability of exploitable vulnerabilities. This on the one hand concerns the organizational security policies that regulate the secure administration and operation of a computer system. On the other hand preventive measures concern the technical aspects within a system's security policy that implement an organizational security policy's strategic decisions. Here we meet with one of the few positive effects of root kits. Having a considerable capacity of finding vulnerabilities, root kits disclose many flaws in design methodology, architecture, paradigms and implementation of contemporary systems that can not be eliminated by short-term measures alone but require reconsideration of fundamental methods of system construction.

Organizational security policies reflect essential strategic decisions for the secure operation of a computer system. Here, the attack technique of root kits implies the consideration of three essential points:

- Root kits exploit errors and weak points in the system software. Security management thus must include a permanent monitoring of the relevant news groups and publications of computer emergency teams such as [6].

- Immediately after discovery of an error the resulting risks must be analyzed. Depending on the result it must be decided whether the error can be tolerated, whether additional security measures will be capable of dealing with the error or whether the affected system functionality must be switched off.
- Generally, the exploration field of root kits must be kept small; servers, demons and open communication ports are the gateways for root kits and must be kept to a minimum.

The implementation of security policies is supported by security models such as [11, 20, 26, 15]. The purpose of a security model is to provide a precise description and analysis of the security properties of a system. Formal descriptions enforce accuracy, uncover mistakes and are within certain limits able to prove the correctness of a security policy's implementation. However, formal techniques require considerable human expertise; apart from applications in closed environments such as the military, security models are not yet common practice.

On a technical level it must be attempted to achieve a high measure of correctness for those system components that constitute the trusted computing base (TCB) [8]. In order to ensure that a system will never execute any root kit program (or any other program from an unknown source) the following preventive measures have to be taken:

- only an authentic, digitally signed operating system is booted on an authentic hardware; a secure boot process is required that verifies signatures of trusted hardware components as well as the signature of the operating system
- only authentic, digitally signed software is executed by the operating system; signatures are checked each time before the operating system's loader loads a program file into memory.

The secure booting technique, secure program loading and digital signatures are long since known. Secure booting has first been described in [12], digital signing algorithms go back e.g. to the work of [25], and the necessary public key infrastructures are long since operational [30, 23]. However, it is not yet common practice to build secure boot modules and loaders into a system's hardware and software.

Intrusion detection systems on the other hand are common practice. Suitable systems must be aware of the techniques of root kits, and they must react very fast. An example are port scan detectors such as *PortSentry* [24]. Deception toolkits such as [7]) simulate a large amount of slow-reacting services at the IP ports and thus considerably slow down a root kit's vulnerability analysis. Tools such as [3] tighten up security by checking for and disabling critical system features. They work through changing permissions and disabling dynamic operating system module loading and try to balance security against usability. Last but not least, a system has to be analyzed systematically and regularly for vulnerabilities. And what better means could there be applied but the analysis components of the root kits themselves...

Finally, we have to point out a very fundamental flaw in the protection schemes of today's operating systems. Root kits gain complete control over a system by assuming the role of "*root*" or "*administrator*". Without the all-embracing privileges of these users, root kits will loose much of their threat. Unfortunately, in many cases such privileges are buried deep within the kernel itself. Several Unix implementations completely circumvent the access right checking within the system call implementation layer whenever the caller has the Unix root user's id "0". Thus on the one hand it is very difficult to restrict *root's* activity sphere in contemporary protection schemes; on the other hand, this protection scheme is coarse-grained to an extend that many services that actually need just a small privilege extension actually run as *root* processes. They thus by far exceed their permission requirements and reduce system security to that of these processes. Again, root kits expose a weakness in today's system design and implementation that cannot be easily eliminated by short-term measures but requires reconsideration of fundamental design issues.

## 4.2 Detection

Because of its very short attack time a root kit attack can either be discovered only automatically (by fast intrusion detection systems, s.a.) or post factum. The multitude and professionalism of the disguising techniques render a coincidental detection virtually impossible. Even an explicit root kit search will provide reliable results only if

- operating system, servers, utilities and application programs are digitally signed
- no key required for signing and signature verification can be accessed by the attacked system
- signatures are checked before any program execution, and checking of the signatures itself is secure; this requires that the operating system itself as the anchor of a chain of trust has been checked and booted by a trustworthy boot procedure [12, 1].

Once such a foundation has been established, additional tools that periodically carry out signature verifications [28] or search for further evidence of a successful root kit attack [29] can be applied.

## 4.3 Neutralization and Recovery

Once a system has been successfully attacked, without the precautions of 4.2 there is no reliable way to discover all backdoors and smoke canisters hiding in modified program files, libraries and operating system modules. The simple reason is that there is a good chance that a program that is used for discovering root kit components may itself be just such a component.

Even an effort to rescue important programs and data bases by removing their storage media (magnetic and optical discs, memory sticks) from the infested system and installing them into a genuine environment cannot lead to success if one of the following conditions is not met:

- the integrity of all files is guarded by a message digest
- the keys used for creating the message digest as well as the finger prints of the genuine original files are kept at a site that physically cannot be accessed from the infected system
- the verification of the message digests is performed in a genuine environment with an authentic operating system and authentic message digest verification programs.

If the attack time is known precisely, without these conditions there is at least a chance to completely recover all file systems from backups. In any other case no save way exists to ever use programs and data files from the infested system again.

# 5 Root Kits and Operating Systems

In order to identify areas where operating systems are essentially involved let us briefly summarize the technical and conceptual weaknesses that are exploited by root kits.

- Root kits exploit errors such as weak type and range checks as well as exotic abstractions (such as */dev/kmem* in Unix systems) and race conditions. They also take advantage of configuration mistakes caused by carelessness, thoughtlessness and the complexity of configuring a networked computer system.

- Although appropriate methods and technologies exist, the authenticity and integrity of operating system, dynamically loaded system modules, servers, applications and libraries in general are not verified when they are loaded into memory.
- Authorization schemes grant all-embracing privileges to system administrators, authorizing one single user account to modify system configuration, program files and operating system all at once.
- Architectural weaknesses are plenty. In monolithic system architectures the lack of isolation between system components encourages the proliferation of corruption caused by a single faulty or corrupted component, including dynamically loaded modules. The correctness verification of critical components such as the trusted computing base, its security models and authentication schemes then is rather pointless.
  The size of the trusted computing base itself as well as its dispersion among various system components are further obstacles that render it virtually impossible to prove implementation correctness.

## 5.1 Programming Errors

Programming errors are neither a new problem nor are they specific to operating systems. Errors exploited by root kits are seldomly located in the more difficult software components dealing for example with parallelism or resource management. In many cases, exploited errors are located in the conventional, sequentially programmed components such as an operating system API or server RPC interface implementation. The fact that seemingly trivial error types nevertheless are quite persistent clearly points out that we are confronted with a fundamental problem of safe execution of erroneous programs that cannot be left to the discipline of software engineering alone. Because programming errors will persist, we will have to accept and deal with their existence.

### 5.1.1 Programming Errors in Trusted Application-Level Software

The protection mechanisms of any multitasking system normally prevent malicious applications from interfering with other applications. Problems arise if trusted applications have extraordinary privileges that neutralize the regular protection mechanisms. Programming errors then can be exploited by an attacker by hijacking the application and bullying it into executing malicious code.

One approach to counteract malicious behavior of trusted processes is to impose strict limits to the sphere of influence of these processes, a problem also known as the confinement problem. Techniques supporting safe execution of untrusted software via dynamic supervision and encapsulation include Java's sandboxing, program shepherding [16], or language-level techniques for safe execution known from dynamically loaded kernel extensions [4, 27, 22, 21] or exokernel modules [9].

These approaches silently accept the discretionary authorization schemes of contemporary mainstream systems, including the notion of an almighty super user. However, many of them have severe drawbacks in development costs, performance and safety. NSA's *Security Enhanced Linux* project [19, 18] pursuits a rather different, more rigorous approach that more deeply considers the reasons for the virtually unlimited proliferation of corruption. Realizing that the fundamental problem are the authorization schemes implemented in contemporary operating system kernels, the Linux kernel was enriched with mandatory access control mechanisms. Section 5.3 will discuss this topic.

### 5.1.2 Programming Errors in Operating Systems

Many programming errors in operating systems that have been exploited by root kits were located in the more trivial parts of the OS code. However, these errors were used to tamper with extremely safety- and security-critical data structures that belong to totally different kernel functions. In order to confine errors

to their local environment and to achieve a correct and robust implementation of security-critical system components, 20 years ago three fundamental architectural rules were developed [8]. These rules (called the three reference monitor principles) demand that access control implementation must be tamperproof, small, and exercise total control on every access operation. Looking for example at a contemporary Linux implementation, we find just the contrary: the access control implementation can be manipulated by every line of system code, is large, and proving the total mediation property is a Sisyphus work.

Errors have, although more rarely, also been discovered in much less frequently used OS parts. Recently, a security hole in a Linux implementation of */dev/kmem* was detected that provided access to a part of the kernel memory. Dynamically loadable modules such as in in Linux or SPIN [4] (where extensions are loaded even into the microkernel) significantly add to the peril of corruption introduction and proliferation. Both features – dynamic kernel extensions and direct access to kernel memory – are examples for a system functionality that is important in research environments but is much less frequently used in commercial environments. A thorough consideration here might show that such functionality is not always required. Avoiding the involved security risks then simply comes down to avoiding unessential and critical system features. Consequently, we clearly need to distinguish between functionally different members of the same system family, the members being compiled and configured to provide only the minimum required functionality.

Microkernel architectures such as [17, 9, 14] supporting fault isolation and downscaling thus are of paramount importance to put a stop to error proliferation.

## 5.2 Code Authenticity and Integrity

From a technical point of view, the problem of validating authenticity and integrity of executable program code is long since solved. The technique is based on digital signatures and certificates that are issued by trustworthy authorities and are checked each time before any program is loaded into memory by a secure validation procedure [12, 1].

Authenticity and integrity checking is built upon a chain of trust, covering the bootstrap-loader (checked by hardware), the operating system (checked by the bootstrap loader) and any application program, library or operating system extension that is loaded into memory.

It is important to notice that this approach does not necessarily depend on extreme approaches like Palladium software or Fritz-chip hardware. In many cases a boot process can be made secure also by administrational means that prevent tampering with hardware and that ensure that the system is always booted from an authenticated read-only medium.

## 5.3 Authorization Schemes

The exploits of root kits point at two major fossils in the authorization schemes of contemporary mainstream operating systems. Firstly, the gate to higher privileges is either completely closed or completely open, and once it is open it provides all-encompassing rights, including rights to modify the operating system, to exchange utility programs and libraries, or to erase entries from log files. Secondly, contemporary authorization schemes lack support for mandatory access control.

### 5.3.1 Almighty Super Users

Contemporary authorization schemes grant all-embracing privileges to system administrators, authorizing one single user account to modify system configuration, program files and operating system all at once. Once a root kit manages to obtain such privileges, it takes over the system completely. Any approach to

eliminate this vulnerability will have to make it more difficult to acquire a higher privilege level and at the time make it less necessary.

Today, several short term solutions exist that address these goals in the context of contemporary access control schemes. In order to make it more difficult to hijack super user privileges, programming error avoidance and isolation techniques as discussed in section 5.1 are applied. In order to make it less necessary, system administrators for example create sandboxes for servers by inventing dummy user accounts, thus misusing a concept originally designed for isolating users on a multiuser system. The success of these short term approaches might be measured by the success of root kits.

A long term approach will have to consider more deeply the reasons for the virtually unlimited proliferation of privileges. It is a disillusioning experience to model even just the basics of the Unix access control scheme in the HRU model [13], a formalism for describing and analyzing the effects of a particular access control policy. Even without including the set-user-id property the resulting model is not monooperational, not monotonic, not monoconditional, and not static. Models of this type in general have undecidable safety properties, or, in other words, do not allow to predict and prove any boundary to privilege proliferation.

The all-or-nothing super user privilege scheme thus is a conceptual weakness, resulting in unbounded privilege proliferation. Its nursing is expensive and – as the long tradition of security incidences and patches shows – far from reliable. In order to put a stop to privilege proliferation, operating systems must provide authorization schemes supporting fine-grained security domains and fine-grained right specifications.

### 5.3.2 Mandatory Access Control

The foundation of security in an operating system is the separation of information. Loscocco et. al. argue in [19, 18] that enforcing and managing separation on the application level requires that an operating system supports mandatory access control (MAC). A powerful argument is that MAC provides the necessary controls to solve the confinement problem mentioned in section 5.1.1. Firstly, MAC counters privilege proliferation caused by permission inheritance in discretionary access control (DAC) schemes. Secondly, MAC provides concepts and mechanisms to define and enforce security domains that restrict the activity sphere of critical applications. Thirdly, MAC provides the foundation for security policies based on labels instead of identities alone. Security policies allow for the implementation of mandatory principles such as need-to-know, need-to-do, or separation of duty, which for example are required to enforce the separation of the right to modify a system component from the right to manipulate a corresponding log file. Labels allow for fine-grained policy decisions based on identity, roles, the trustworthiness of a program, or the confidentiality of a file.

In DAC access decisions are based on ownership of objects and identity of subjects alone. Individual users exercise complete control over their objects, making it impossible to enforce a system-wide mandatory security policy. Consequently, MAC is no new abstraction level that can be built on top of DAC; it is a fundamentally different concept that must be anchored within the operating system kernel itself.

# 6 Summary

The starting point of this paper was a survey of the works of root kits. We discussed their techniques for analysis and exploitation of system weaknesses, concealment techniques, and backdoor implantation. We also discussed attack prevention, discovery, and recovery.

Root kits expose several flaws in the design and implementation of contemporary operating systems which cannot be easily eliminated by short-term measures. Firstly, root kits exploit implementation flaws that are re-introduced again and again by software updates and new system functionality. Secondly, they exploit conceptual weaknesses in contemporary authorization schemes and monolithic system architectures, the former being unable to prevent privilege proliferation, the latter being unable to prevent

error proliferation. Thirdly, concealment techniques work well because operating systems naively assume "trusted" applications to be error-free and to not be hijacked, and trust that operating system code as well as utilities, servers, and applications has not been tampered with. In order to deal with these flaws and misconceptions a reconsideration of fundamental system design principles is required with respect to functional as well as quality properties.

Firstly, in order to reduce the probability of erroneous system code, systems must allow for a fine-grained functionality scaling. Secondly, the fundamental kernel security mechanisms must be powerful enough to support a range of higher-level security abstractions, including security domains and security policies, the former supporting application-level error isolation and privilege encapsulation, the latter supporting DAC and MAC security.

Because the low-level kernel security mechanisms constitute the critical foundation of the entire trusted computing base, high quality requirements with respect to their implementation apply. Firstly, the mechanisms and their implementation have to be simple enough to be described and analyzed by a formal security model. Then, the implementation must be provable correct, tamperproof, and provide total control. This implies that the implementation is very small, encapsulated and located at an architectural level that controls any communication between security-relevant entities. In other words, the implementation of the security core must suffice the three reference monitor principles.

Monolithic OS kernel architectures are incapable of realizing such requirements, because they cannot be safely and efficiently scaled to the required functionality, lack safe and efficient error isolation mechanisms and thus would have to be proven correct as a whole.

Microkernel architectures do profoundly better. While, of course, microkernels provide architectural support for kernel-level error isolation and functional scalability, the security core can be implemented within a small microkernel in order to establish a secure and tamperproof foundation for implementing higher-level security abstractions. Its formal definition, proven safety, correctness and tamperproofness is the foundation for proving corresponding quality properties of higher abstraction levels, thus in correspondence to a functional hierarchy forms a quality hierarchy that allows to argue about the quality properties of the entire trusted computing base.

Many approaches that address security flaws have been developed within the OS research community, but until today, only a few have been adopted by commercial OS providers [10]. If the awareness of fortresses built upon sand [2] had proliferated in the commercial world as fast as errors and privileges in our mainstream operating systems do, root kits would no longer exist and the relevance of this article would have long since ceased.

# References


[1] Bill Arbaugh, Dave Farber, and Jonathan Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 65–71. IEEE, 1997.

[2] D. Baker. Fortresses Built Upon Sand. In *Proceedings of the New Security Paradigms Workshop*. ACM SIG on Security, Audit and Control, ACM Press, 1996.

[3] http://www.bastille-linux.org, 2003.

[4] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[5] Bob Blakley. The Emperor's Old Armor. In *Proceedings of the New Security Paradigms Workshop*, pages 2–16. ACM SIG on Security, Audit and Control, ACM Press, 1996.

[6] CERT Coordination Center, 2003. http://www.cert.org.

[7] Fred Cohen and Associates. Deception Toolkit, 2003. http://all.net/dtk.

[8] Department of Defense. *Trusted Computer System Evaluation Criteria*, August 1983.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[10] Dawson R. Engler and M. Frans Kaashoek. Exterminate All Operating System Abstractions. In *Proceedings of the Fifth IEEE Workshop on Hot Topics in Operating Systems*, pages 78–85, 1995.

[11] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, April 1982.

[12] Michael Gross. Vertrauenswürdiges Booten als Grunglage authentischer Basissysteme. In A. Pfitzmann and E. Raubold, editors, *VIS'91 — Verlässliche Informationssysteme*, pages 190–207. Springer Verlag, March 1991. ISBN 3-540-53911-5.

[13] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. On Protection in Operating Systems. *Operating Systems Review, special issue for the 5th Symposium on Operating Systems Principles*, 9(5):14–24, November 1975.

[14] Trent Jaeger, Jochen Liedtke, Vsevolod Panteleenko, Yoonho Park, and Nayeem Islam. Security Architecture for Component-based Operating Systems. In *8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[15] Günther Karjoth, Danny B. Lange, and Mitsuru Oshima. A Security Model for Aglets. In *Mobile Agents Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 188–205. Springer Verlag, 1998.

[16] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, San Francisco, CA, 2002. USENIX. ISBN 1-931971-00-5.

[17] Jochen Liedtke. On $\mu$-Kernel Construction. *Operating Systems Review. Special issue for the Fifteenth ACM Symposium on Operating System Principles*, 29(5):237–250, December 1995.

[18] Peter A. Loscocco and Stephen D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In Clem Cole, editor, *Proceedings of the FREENIX Track, 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, pages 29–42. USENIX, 2001.

[19] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.

[20] Teresa F. Lunt, Dorothy E. Denning, Roger R. Schell, Mark Heckmann, and William R. Shockley. The SeaView Security Model. *IEEE Transactions on Software Engineering*, 16(6):593–607, June 1990.

[21] George C. Necula and Peter Lee. Safe Kernel Extensions Without Runtime Checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[22] P. Pardyak and Brian N. Bershad. Dynamic Binding for an Extensible System. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996. Published in ACM SIGOPS Operating Systems Review, Vol 30, Oct. 1996, pages 201–212.

[23] PKIX Working Group. Internet X.509 PKI: Roadmap, July 2002. http://www.ietf.org/ietf/1id-abstracts.txt (Oct. 2002).

[24] Psionic Technologies. Port Sentry Version 2.0b1, 2002. http://www.psionic.com.

[25] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[26] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[27] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, 1996.

[28] Tripwire. The Tripwire Open Source Project, 2003. http://www.tripwire.org.

[29] Vancouver Pages. Root Kit Detectors, 2003. http://www.vancouver-webpages.com/rkdet.

[30] ITU-T Recommendation X.509, June 1997.