NAME

security - introduction to security under FreeBSD

DESCRIPTION

Security is a function that begins and ends with the system administrator. While all BSD multi-user systems have some inherent security, the job of building and maintaining additional security mechanisms to keep users "honest" is probably one of the single largest undertakings of the sysadmin. Machines are only as secure as you make them, and security concerns are ever competing with the human necessity for convenience. UNIX systems, in general, are capable of running a huge number of simultaneous processes and many of these processes operate as servers -- meaning that external entities can connect and talk to them. As yesterday's mini-computers and mainframes become today's desktops, and as computers become networked and internetworked, security becomes an ever bigger issue.

Security is best implemented through a layered onion approach. In a nutshell, what you want to do is to create as many layers of security as are convenient and then carefully monitor the system for intrusions.

System security also pertains to dealing with various forms of attacks, including attacks that attempt to crash or otherwise make a system unusable but do not attempt to break root. Security concerns can be split up into several categories:

1. Denial of Service attacks (DoS)
2. User account compromises
3. Root compromise through accessible servers
4. Root compromise via user accounts
5. Backdoor creation

A denial of service attack is an action that deprives the machine of needed resources. Typically, DoS attacks are brute-force mechanisms that attempt to crash or otherwise make a machine unusable by overwhelming its servers or network stack. Some DoS attacks try to take advantages of bugs in the networking stack to crash a machine with a single packet. The latter can only be fixed by applying a bug fix to the kernel. Attacks on servers can often be fixed by properly specifying options to limit the load the servers incur on the system under adverse conditions. Brute-force network attacks are harder to deal with. A spoofed-packet attack, for example, is nearly impossible to stop short of cutting your system off from the Internet. It may not be able to take your machine down, but it can fill up your Internet pipe.

A user account compromise is even more common than a DoS attack. Some sysadmins still run **telnetd**

and ftpd(8) servers on their machines. These servers, by default, do not operate over encrypted connections. The result is that if you have any moderate-sized user base, one or more of your users logging into your system from a remote location (which is the most common and convenient way to log in to a system) will have his or her password sniffed. The attentive system administrator will analyze his remote access logs looking for suspicious source addresses even for successful logins.

One must always assume that once an attacker has access to a user account, the attacker can break root. However, the reality is that in a well secured and maintained system, access to a user account does not necessarily give the attacker access to root. The distinction is important because without access to root the attacker cannot generally hide his tracks and may, at best, be able to do nothing more than mess with the user's files or crash the machine. User account compromises are very common because users tend not to take the precautions that sysadmins take.

System administrators must keep in mind that there are potentially many ways to break root on a machine. The attacker may know the root password, the attacker may find a bug in a root-run server and be able to break root over a network connection to that server, or the attacker may know of a bug in an SUID-root program that allows the attacker to break root once he has broken into a user's account. If an attacker has found a way to break root on a machine, the attacker may not have a need to install a backdoor. Many of the root holes found and closed to date involve a considerable amount of work by the attacker to clean up after himself, so most attackers do install backdoors. This gives you a convenient way to detect the attacker. Making it impossible for an attacker to install a backdoor may actually be detrimental to your security because it will not close off the hole the attacker used to break in originally.

Security remedies should always be implemented with a multi-layered "onion peel" approach and can be categorized as follows:

1. Securing root and staff accounts
2. Securing root -- root-run servers and SUID/SGID binaries
3. Securing user accounts
4. Securing the password file
5. Securing the kernel core, raw devices, and file systems
6. Quick detection of inappropriate changes made to the system
7. Paranoia

SECURING THE ROOT ACCOUNT AND SECURING STAFF ACCOUNTS

Do not bother securing staff accounts if you have not secured the root account. Most systems have a password assigned to the root account. The first thing you do is assume that the password is *always* compromised. This does not mean that you should remove the password. The password is almost always necessary for console access to the machine. What it does mean is that you should not make it possible to use the password outside of the console or possibly even with a su(1) utility. For example, make sure that your PTYs are specified as being "insecure" in the */etc/ttys* file so that direct root logins via telnet(1) are disallowed. If using other login services such as sshd(8), make sure that direct root logins are disabled there as well. Consider every access method -- services such as ftp(1) often fall through the cracks. Direct root logins should only be allowed via the system console.

Of course, as a sysadmin you have to be able to get to root, so we open up a few holes. But we make sure these holes require additional password verification to operate. One way to make root accessible is to add appropriate staff accounts to the "wheel" group (in */etc/group*). The staff members placed in the wheel group are allowed to su(1) to root. You should never give staff members native wheel access by putting them in the wheel group in their password entry. Staff accounts should be placed in a "staff" group, and then added to the wheel group via the */etc/group* file. Only those staff members who actually need to have root access should be placed in the wheel group. It is also possible, when using an authentication method such as Kerberos, to use Kerberos's *.k5login* file in the root account to allow a ksu(1) to root without having to place anyone at all in the wheel group. This may be the better solution since the wheel mechanism still allows an intruder to break root if the intruder has gotten hold of your password file and can break into a staff account. While having the wheel mechanism is better than having nothing at all, it is not necessarily the safest option.

An indirect way to secure the root account is to secure your staff accounts by using an alternative login access method and *'ing out the crypted password for the staff accounts. This way an intruder may be able to steal the password file but will not be able to break into any staff accounts or root, even if root has a crypted password associated with it (assuming, of course, that you have limited root access to the console). Staff members get into their staff accounts through a secure login mechanism such as kerberos(8) or ssh(1) using a private/public key pair. When you use something like Kerberos you generally must secure the machines which run the Kerberos servers and your desktop workstation. When you use a public/private key pair with SSH, you must generally secure the machine you are logging in *from* (typically your workstation), but you can also add an additional layer of protection to the key pair by password protecting the keypair when you create it with ssh-keygen(1). Being able to star-out the passwords for staff accounts also guarantees that staff members can only log in through secure access methods that you have set up. You can thus force all staff members to use secure, encrypted connections for all their sessions which closes an important hole used by many intruders: that of sniffing the network from an unrelated, less secure machine.

The more indirect security mechanisms also assume that you are logging in from a more restrictive

server to a less restrictive server. For example, if your main box is running all sorts of servers, your workstation should not be running any. In order for your workstation to be reasonably secure you should run as few servers as possible, up to and including no servers at all, and you should run a password-protected screen blanker. Of course, given physical access to a workstation, an attacker can break any sort of security you put on it. This is definitely a problem that you should consider but you should also consider the fact that the vast majority of break-ins occur remotely, over a network, from people who do not have physical access to your workstation or servers.

Using something like Kerberos also gives you the ability to disable or change the password for a staff account in one place and have it immediately affect all the machines the staff member may have an account on. If a staff member's account gets compromised, the ability to instantly change his password on all machines should not be underrated. With discrete passwords, changing a password on N machines can be a mess. You can also impose re-passwording restrictions with Kerberos: not only can a Kerberos ticket be made to timeout after a while, but the Kerberos system can require that the user choose a new password after a certain period of time (say, once a month).

SECURING ROOT -- ROOT-RUN SERVERS AND SUID/SGID BINARIES

The prudent sysadmin only runs the servers he needs to, no more, no less. Be aware that third party servers are often the most bug-prone. For example, running an old version of imapd(8) or popper(8) (*ports/mail/popper*) is like giving a universal root ticket out to the entire world. Never run a server that you have not checked out carefully. Many servers do not need to be run as root. For example, the talkd(8), comsat(8), and fingerd(8) daemons can be run in special user "sandboxes". A sandbox is not perfect unless you go to a large amount of trouble, but the onion approach to security still stands: if someone is able to break in through a server running in a sandbox, they still have to break out of the sandbox. The more layers the attacker must break through, the lower the likelihood of his success. Root holes have historically been found in virtually every server ever run as root, including basic system servers. If you are running a machine through which people only log in via sshd(8) and never log in via **telnetd** then turn off this service!

FreeBSD now defaults to running talkd(8), comsat(8), and fingerd(8) in a sandbox. Depending on whether you are installing a new system or upgrading an existing system, the special user accounts used by these sandboxes may not be installed. The prudent sysadmin would research and implement sandboxes for servers whenever possible.

There are a number of other servers that typically do not run in sandboxes: sendmail(8), popper(8), imapd(8), ftpd(8), and others. There are alternatives to some of these, but installing them may require more work than you are willing to put (the convenience factor strikes again). You may have to run these servers as root and rely on other mechanisms to detect break-ins that might occur through them.

The other big potential root hole in a system are the SUID-root and SGID binaries installed on the

system. Most of these binaries, such as su(1), reside in */bin*, */sbin*, */usr/bin*, or */usr/sbin*. While nothing is 100% safe, the system-default SUID and SGID binaries can be considered reasonably safe. Still, root holes are occasionally found in these binaries. A root hole was found in Xlib in 1998 that made xterm(1) (*ports/x11/xterm*) (which is typically SUID) vulnerable. It is better to be safe than sorry and the prudent sysadmin will restrict SUID binaries that only staff should run to a special group that only staff can access, and get rid of ("chmod 000") any SUID binaries that nobody uses. A server with no display generally does not need an xterm(1) (*ports/x11/xterm*) binary. SGID binaries can be almost as dangerous. If an intruder can break an SGID-kmem binary the intruder might be able to read */dev/kmem* and thus read the crypted password file, potentially compromising any passworded account. Alternatively an intruder who breaks group "kmem" can monitor keystrokes sent through PTYs, including PTYs used by users who log in through secure methods. An intruder that breaks the "tty" group can write to almost any user's TTY. If a user is running a terminal program or emulator with a keyboard-simulation feature, the intruder can potentially generate a data stream that causes the user's terminal to echo a command, which is then run as that user.

SECURING USER ACCOUNTS

User accounts are usually the most difficult to secure. While you can impose draconian access restrictions on your staff and *-out their passwords, you may not be able to do so with any general user accounts you might have. If you do have sufficient control then you may win out and be able to secure the user accounts properly. If not, you simply have to be more vigilant in your monitoring of those accounts. Use of SSH and Kerberos for user accounts is more problematic due to the extra administration and technical support required, but still a very good solution compared to a crypted password file.

SECURING THE PASSWORD FILE

The only sure fire way is to *-out as many passwords as you can and use SSH or Kerberos for access to those accounts. Even though the crypted password file (*/etc/spwd.db*) can only be read by root, it may be possible for an intruder to obtain read access to that file even if the attacker cannot obtain root-write access.

Your security scripts should always check for and report changes to the password file (see *CHECKING FILE INTEGRITY* below).

SECURING THE KERNEL CORE, RAW DEVICES, AND FILE SYSTEMS

If an attacker breaks root he can do just about anything, but there are certain conveniences. For example, most modern kernels have a packet sniffing device driver built in. Under FreeBSD it is called the bpf(4) device. An intruder will commonly attempt to run a packet sniffer on a compromised machine. You do not need to give the intruder the capability and most systems should not have the bpf(4) device compiled in.

But even if you turn off the bpf(4) device, you still have */dev/mem* and */dev/kmem* to worry about. For that matter, the intruder can still write to raw disk devices. Also, there is another kernel feature called the module loader, kldload(8). An enterprising intruder can use a KLD module to install his own bpf(4) device or other sniffing device on a running kernel. To avoid these problems you have to run the kernel at a higher security level, at least level 1. The security level can be set with a sysctl(8) on the *kern.securelevel* variable. Once you have set the security level to 1, write access to raw devices will be denied and special chflags(1) flags, such as **schg**, will be enforced. You must also ensure that the **schg** flag is set on critical startup binaries, directories, and script files -- everything that gets run up to the point where the security level is set. This might be overdoing it, and upgrading the system is much more difficult when you operate at a higher security level. You may compromise and run the system at a higher security level but not set the **schg** flag for every system file and directory under the sun. Another possibility is to simply mount */* and */usr* read-only. It should be noted that being too draconian in what you attempt to protect may prevent the all-important detection of an intrusion.

The kernel runs with five different security levels. Any super-user process can raise the level, but no process can lower it. The security levels are:

-1 Permanently insecure mode - always run the system in insecure mode. This is the default initial value.

0 Insecure mode - immutable and append-only flags may be turned off. All devices may be read or written subject to their permissions.

1 Secure mode - the system immutable and system append-only flags may not be turned off; disks for mounted file systems, */dev/mem* and */dev/kmem* may not be opened for writing; */dev/io* (if your platform has it) may not be opened at all; kernel modules (see kld(4)) may not be loaded or unloaded. The kernel debugger may not be entered using the *debug.kdb.enter* sysctl unless a MAC(9) policy grants access, for example using mac_ddb(4). A panic or trap cannot be forced using the *debug.kdb.panic*, *debug.kdb.panic_str* and other sysctl’s.

2 Highly secure mode - same as secure mode, plus disks may not be opened for writing (except by mount(2)) whether mounted or not. This level precludes tampering with file systems by unmounting them, but also inhibits running newfs(8) while the system is multi-user.

In addition, kernel time changes are restricted to less than or equal to one second. Attempts to change the time by more than this will log the message "Time adjustment clamped to +1 second".

3 Network secure mode - same as highly secure mode, plus IP packet filter rules (see ipfw(8), ipfirewall(4) and pfctl(8)) cannot be changed and dummynet(4) or pf(4) configuration cannot be adjusted.

The security level can be configured with variables documented in rc.conf(5).

CHECKING FILE INTEGRITY: BINARIES, CONFIG FILES, ETC

When it comes right down to it, you can only protect your core system configuration and control files so much before the convenience factor rears its ugly head. For example, using chflags(1) to set the **schg** bit on most of the files in */* and */usr* is probably counterproductive because while it may protect the files, it also closes a detection window. The last layer of your security onion is perhaps the most important -- detection. The rest of your security is pretty much useless (or, worse, presents you with a false sense of safety) if you cannot detect potential incursions. Half the job of the onion is to slow down the attacker rather than stop him in order to give the detection layer a chance to catch him in the act.

The best way to detect an incursion is to look for modified, missing, or unexpected files. The best way to look for modified files is from another (often centralized) limited-access system. Writing your security scripts on the extra-secure limited-access system makes them mostly invisible to potential attackers, and this is important. In order to take maximum advantage you generally have to give the limited-access box significant access to the other machines in the business, usually either by doing a read-only NFS export of the other machines to the limited-access box, or by setting up SSH keypairs to allow the limit-access box to SSH to the other machines. Except for its network traffic, NFS is the least visible method -- allowing you to monitor the file systems on each client box virtually undetected. If your limited-access server is connected to the client boxes through a switch, the NFS method is often the better choice. If your limited-access server is connected to the client boxes through a hub or through several layers of routing, the NFS method may be too insecure (network-wise) and using SSH may be the better choice even with the audit-trail tracks that SSH lays.

Once you give a limit-access box at least read access to the client systems it is supposed to monitor, you must write scripts to do the actual monitoring. Given an NFS mount, you can write scripts out of simple system utilities such as find(1) and md5(1). It is best to physically md5(1) the client-box files boxes at least once a day, and to test control files such as those found in */etc* and */usr/local/etc* even more often. When mismatches are found relative to the base MD5 information the limited-access machine knows is valid, it should scream at a sysadmin to go check it out. A good security script will also check for inappropriate SUID binaries and for new or deleted files on system partitions such as */* and */usr*.

When using SSH rather than NFS, writing the security script is much more difficult. You essentially have to scp(1) the scripts to the client box in order to run them, making them visible, and for safety you also need to scp(1) the binaries (such as find(1)) that those scripts use. The sshd(8) daemon on the client box may already be compromised. All in all, using SSH may be necessary when running over unsecure links, but it is also a lot harder to deal with.

A good security script will also check for changes to user and staff members access configuration files: *.rhosts*, *.shosts*, *.ssh/authorized_keys* and so forth, files that might fall outside the purview of the MD5

check.

If you have a huge amount of user disk space it may take too long to run through every file on those partitions. In this case, setting mount flags to disallow SUID binaries on those partitions is a good idea. The **nosuid** option (see mount(8)) is what you want to look into. I would scan them anyway at least once a week, since the object of this layer is to detect a break-in whether or not the break-in is effective.

Process accounting (see accton(8)) is a relatively low-overhead feature of the operating system which I recommend using as a post-break-in evaluation mechanism. It is especially useful in tracking down how an intruder has actually broken into a system, assuming the file is still intact after the break-in occurs.

Finally, security scripts should process the log files and the logs themselves should be generated in as secure a manner as possible -- remote syslog can be very useful. An intruder tries to cover his tracks, and log files are critical to the sysadmin trying to track down the time and method of the initial break-in. One way to keep a permanent record of the log files is to run the system console to a serial port and collect the information on a continuing basis through a secure machine monitoring the consoles.

PARANOIA

A little paranoia never hurts. As a rule, a sysadmin can add any number of security features as long as they do not affect convenience, and can add security features that do affect convenience with some added thought. Even more importantly, a security administrator should mix it up a bit -- if you use recommendations such as those given by this manual page verbatim, you give away your methodologies to the prospective attacker who also has access to this manual page.

SPECIAL SECTION ON DoS ATTACKS

This section covers Denial of Service attacks. A DoS attack is typically a packet attack. While there is not much you can do about modern spoofed packet attacks that saturate your network, you can generally limit the damage by ensuring that the attacks cannot take down your servers.

1. Limiting server forks

2. Limiting springboard attacks (ICMP response attacks, ping broadcast, etc.)

3. Kernel Route Cache

A common DoS attack is against a forking server that attempts to cause the server to eat processes, file descriptors, and memory until the machine dies. The inetd(8) server has several options to limit this sort of attack. It should be noted that while it is possible to prevent a machine from going down it is not generally possible to prevent a service from being disrupted by the attack. Read the inetd(8) manual page carefully and pay specific attention to the **-c**, **-C**, and **-R** options. Note that spoofed-IP attacks will

circumvent the **-C** option to inetd(8), so typically a combination of options must be used. Some standalone servers have self-fork-limitation parameters.

The sendmail(8) daemon has its **-OMaxDaemonChildren** option which tends to work much better than trying to use sendmail(8)'s load limiting options due to the load lag. You should specify a *MaxDaemonChildren* parameter when you start sendmail(8) high enough to handle your expected load but not so high that the computer cannot handle that number of **sendmail**'s without falling on its face. It is also prudent to run sendmail(8) in "queued" mode (**-ODeliveryMode=queued**) and to run the daemon ("**sendmail -bd**") separate from the queue-runs ("**sendmail -q15m**"). If you still want real-time delivery you can run the queue at a much lower interval, such as **-q1m**, but be sure to specify a reasonable *MaxDaemonChildren* option for that sendmail(8) to prevent cascade failures.

The syslogd(8) daemon can be attacked directly and it is strongly recommended that you use the **-s** option whenever possible, and the **-a** option otherwise.

You should also be fairly careful with connect-back services such as tcpwrapper's reverse-identd, which can be attacked directly. You generally do not want to use the reverse-ident feature of tcpwrappers for this reason.

It is a very good idea to protect internal services from external access by firewalling them off at your border routers. The idea here is to prevent saturation attacks from outside your LAN, not so much to protect internal services from network-based root compromise. Always configure an exclusive firewall, i.e., 'firewall everything *except* ports A, B, C, D, and M-Z'. This way you can firewall off all of your low ports except for certain specific services such as talkd(8), sendmail(8), and other internet-accessible services. If you try to configure the firewall the other way -- as an inclusive or permissive firewall, there is a good chance that you will forget to "close" a couple of services or that you will add a new internal service and forget to update the firewall. You can still open up the high-numbered port range on the firewall to allow permissive-like operation without compromising your low ports. Also take note that FreeBSD allows you to control the range of port numbers used for dynamic binding via the various *net.inet.ip.portrange* sysctl's ("sysctl net.inet.ip.portrange"), which can also ease the complexity of your firewall's configuration. I usually use a normal first/last range of 4000 to 5000, and a hiport range of 49152 to 65535, then block everything under 4000 off in my firewall (except for certain specific internet-accessible ports, of course).

Another common DoS attack is called a springboard attack -- to attack a server in a manner that causes the server to generate responses which then overload the server, the local network, or some other machine. The most common attack of this nature is the ICMP PING BROADCAST attack. The attacker spoofs ping packets sent to your LAN's broadcast address with the source IP address set to the actual machine they wish to attack. If your border routers are not configured to stomp on ping's to broadcast addresses, your LAN winds up generating sufficient responses to the spoofed source address

to saturate the victim, especially when the attacker uses the same trick on several dozen broadcast addresses over several dozen different networks at once. Broadcast attacks of over a hundred and twenty megabits have been measured. A second common springboard attack is against the ICMP error reporting system. By constructing packets that generate ICMP error responses, an attacker can saturate a server's incoming network and cause the server to saturate its outgoing network with ICMP responses. This type of attack can also crash the server by running it out of *mbuf*'s, especially if the server cannot drain the ICMP responses it generates fast enough. The FreeBSD kernel has a new kernel compile option called ICMP_BANDLIM which limits the effectiveness of these sorts of attacks. The last major class of springboard attacks is related to certain internal inetd(8) services such as the UDP echo service. An attacker simply spoofs a UDP packet with the source address being server A's echo port, and the destination address being server B's echo port, where server A and B are both on your LAN. The two servers then bounce this one packet back and forth between each other. The attacker can overload both servers and their LANs simply by injecting a few packets in this manner. Similar problems exist with the internal chargen port. A competent sysadmin will turn off all of these inetd(8)-internal test services.

ACCESS ISSUES WITH KERBEROS AND SSH

There are a few issues with both Kerberos and SSH that need to be addressed if you intend to use them. Kerberos5 is an excellent authentication protocol but the kerberized telnet(1) suck rocks. There are bugs that make them unsuitable for dealing with binary streams. Also, by default Kerberos does not encrypt a session unless you use the **-x** option. SSH encrypts everything by default.

SSH works quite well in every respect except when it is set up to forward encryption keys. What this means is that if you have a secure workstation holding keys that give you access to the rest of the system, and you ssh(1) to an unsecure machine, your keys become exposed. The actual keys themselves are not exposed, but ssh(1) installs a forwarding port for the duration of your login and if an attacker has broken root on the unsecure machine he can utilize that port to use your keys to gain access to any other machine that your keys unlock.

We recommend that you use SSH in combination with Kerberos whenever possible for staff logins. SSH can be compiled with Kerberos support. This reduces your reliance on potentially exposable SSH keys while at the same time protecting passwords via Kerberos. SSH keys should only be used for automated tasks from secure machines (something that Kerberos is unsuited to). We also recommend that you either turn off key-forwarding in the SSH configuration, or that you make use of the *from=IP/DOMAIN* option that SSH allows in its *authorized_keys* file to make the key only usable to entities logging in from specific machines.

KNOBS AND TWEAKS

FreeBSD provides several knobs and tweak handles that make some introspection information access more restricted. Some people consider this as improving system security, so the knobs are briefly listed there, together with controls which enable some mitigations of the hardware state leaks.

Hardware mitigation sysctl knobs described below have been moved under *machdep.mitigations*, with backwards-compatibility shims to accept the existing names. A future change will rationalize the sense of the individual sysctls (so that enabled / true always indicates that the mitigation is active). For that reason the previous names remain the canonical way to set the mitigations, and are documented here. Backwards compatibility shims for the interim sysctls under *machdep.mitigations* will not be added.

security.bsd.see_other_uids	Controls visibility and reachability of subjects (e.g., processes) and objects (e.g., sockets) owned by a different uid. The knob directly affects the kern.proc sysctls filtering of data, which results in restricted output from utilities like ps(1).
security.bsd.see_other_gids	Same, for subjects and objects owned by a different gid.
security.bsd.see_jail_proc	Same, for subjects and objects belonging to a different jail, including sub-jails.
security.bsd.conservative_signals	When enabled, unprivileged users are only allowed to send job control and usual termination signals like SIGKILL, SIGINT, and SIGTERM, to the processes executing programs with changed uids.
security.bsd.unprivileged_proc_debug	Controls availability of the process debugging facilities to non-root users. See also proccontrol(1) mode trace.
vm.pmap.pti	Tunable, amd64-only. Enables mode of operation of virtual memory system where usermode page tables are sanitized to prevent so-called Meltdown information leak on some Intel CPUs. By default, the system detects whether the CPU needs the workaround, and enables it automatically. See also proccontrol(1) mode kpti.
machdep.mitigations.flush_rsb_ctxsw	amd64. Controls Return Stack Buffer flush on context switch, to prevent cross-process ret2spec attacks. Only needed, and only enabled by default, if the machine supports SMEP, otherwise IBRS would do necessary flushing on kernel entry anyway.
hw.mds_disable	amd64 and i386. Controls Microarchitectural Data Sampling hardware information leak mitigation.
hw.spec_store_bypass_disable	amd64 and i386. Controls Speculative Store Bypass hardware

	information leak mitigation.
hw.ibrs_disable	amd64 and i386. Controls Indirect Branch Restricted Speculation hardware information leak mitigation.
machdep.syscall_ret_flush_l1d	amd64. Controls force-flush of L1D cache on return from syscalls which report errors other than EEXIST, EAGAIN, EXDEV, ENOENT, ENOTCONN, and EINPROGRESS. This is mostly a paranoid setting added to prevent hypothetical exploitation of unknown gadgets for unknown hardware issues. The error codes exclusion list is composed of the most common errors which typically occurs on normal system operation.
machdep.nmi_flush_l1d_sw	amd64. Controls force-flush of L1D cache on NMI; this provides software assist for bhyve mitigation of L1 terminal fault hardware information leak.
hw.vmm.vmx.l1d_flush	amd64. Controls the mitigation of L1 Terminal Fault in bhyve hypervisor.
vm.pmap.allow_2m_x_ept	amd64. Allows the use of superpages for executable mappings under the EPT page table format used by hypervisors on Intel CPUs to map the guest physical address space to machine physical memory. May be disabled to work around a CPU Erratum called Machine Check Error Avoidance on Page Size Change.
machdep.mitigations.rngds.enable	amd64 and i386. Controls mitigation of Special Register Buffer Data Sampling versus optimization of the MCU access. When set to zero, the mitigation is disabled, and the RDSEED and RDRAND instructions do not incur serialization overhead for shared buffer accesses, and do not serialize off-core memory accessses.
kern.elf32.aslr.enable	Controls system-global Address Space Layout Randomization (ASLR) for normal non-PIE (Position Independent Executable) 32-bit ELF binaries. See also the proccontrol(1) aslr mode, also affected by the per-image control note flag.
kern.elf32.aslr.pie_enable	Controls system-global Address Space Layout Randomization

	for position-independent (PIE) 32-bit binaries.
kern.elf32.aslr.honor_sbrk	Makes ASLR less aggressive and more compatible with old binaries relying on the sbrk area.
kern.elf32.aslr.stack	If ASLR is enabled for a binary, a non-zero value enables randomization of the stack. Otherwise, the stack is mapped at a fixed location determined by the process ABI.
kern.elf64.aslr.enable	ASLR control for 64-bit ELF binaries.
kern.elf64.aslr.pie_enable	ASLR control for 64-bit ELF PIEs.
kern.elf64.aslr.honor_sbrk	ASLR sbrk compatibility control for 64-bit binaries.
kern.elf64.aslr.stack	Controls stack address randomization for 64-bit binaries.
kern.elf32.nxstack	Enables non-executable stack for 32-bit processes. Enabled by default if supported by hardware and corresponding binary.
kern.elf64.nxstack	Enables non-executable stack for 64-bit processes.
kern.elf32.allow_wx	Enables mapping of simultaneously writable and executable pages for 32-bit processes.
kern.elf64.allow_wx	Enables mapping of simultaneously writable and executable pages for 64-bit processes.

SEE ALSO

chflags(1), find(1), md5(1), netstat(1), openssl(1), proccontrol(1), ps(1), ssh(1), xdm(1) (*ports/x11/xorg-clients*), group(5), ttys(5), mitigations(7), accton(8), init(8), sshd(8), sysctl(8), syslogd(8), vipw(8)

HISTORY

The **security** manual page was originally written by Matthew Dillon and first appeared in FreeBSD 3.1, December 1998.