Building Servers as Appliances for Improved Security

Algis Kibirkstis

Building Servers as Appliances for Improved Security

GIAC (GSEC) Gold Certification

Author: Algis Kibirkstis, kibia@ethisecure.com
Advisor: Egan Hadsell

Accepted: February 16, 2010

Abstract

Installing, configuring and maintaining hardened servers are core components of a defense-in-depth strategy when protecting computing infrastructure. A common hardening tactic is to disable unnecessary features, functions and capabilities; the underlying problem with this tactic is that dormant vulnerabilities can be awoken by simply re-enabling those services. Stripping down servers, through the minimization of bloated operating system platforms, is an effective means to counteract the possibility of enabling unnecessary or undesirable services – they are simply not installed. Commercial network appliances based on UNIX variants, such as load balancers and intrusion detection systems, continue to be deployed on minimized platforms to not only limit potential vulnerabilities, but also to improve system performance and reduce the need to patch. So if minimization is an effective means of hardening network appliances, shouldn't the same tactic be used when deploying servers? This paper will present minimization as a fundamental tactic when deploying hardened servers based on a popular Linux platform (CentOS/VM), and propose a methodology for identifying core functions and discovering necessary software dependencies.

1. Introduction

Defense-in-Depth is a term commonly used when describing a layered model for protecting computing environments; by having multiple layers of protection, from the perimeter of the network to each computing system at the core, security-related failures at any single layer should not compromise the confidentiality, integrity or availability of the overall system. In this day and age, simple reliance on firewalls for protecting is generally considered to be imprudent (Brining, 2008), for they offer no network-level protection in case of failure, poor configuration, software misbehavior, or unauthorized access attempts posing as legitimate traffic; nor can they offer any protection if communications circumvent the firewall itself.

So in order to move beyond a primitive 1990's security model (Avolio, 2009) of a hard and crunchy outside (a firewall) protecting a soft and chewy inside (the network infrastructure worth protecting), what other mechanisms are available to support Defense-in-Depth (Northcutt, 2007)? Looking inside from the perimeter, popular security strategies include multiple packet filtering mechanisms with DMZ buffer zones in-between, perhaps some application-level firewalls or proxy server implementations for sensitive interfaces, maybe some remote access protection mechanisms, plus network intrusion protection and intrusion detection systems (NIPS/NIDS). In the event that undesirable traffic is able to penetrate all those network-based defenses and reach a system's network interface, the last line of defense would be host-level protection mechanisms.

Such last-line defenses can be enhanced with other technologies such as host intrusion detection and intrusion protection systems (HIDS/HIPS) plus host firewall implementations. At the very core is the target system of interest, perhaps a web server vulnerable to a denial-of-service attack or a database server containing credit card numbers, itself hardened through prudent configuration of its operating system (OS), applications, services and access controls.

A classic strategy used to perform host-level hardening is to turn off network services that aren't required for a target system's business functions – for if a service is

not made available, any vulnerabilities with that service (past, present or future) known, cannot be exploited. But is it as simple as that? “Is that all there is?” (Leiber & Stoller, 1968) Is simple disabling sufficient to eradicate host-level threats against such vulnerable services?

The answer is a resounding “no”. Even with the most stringent policies in place to control user behavior, disabled functionalities can be quickly brought back online by system administrators accustomed to using specific tools when performing updates or troubleshooting tasks. While the risks associated with temporary enabling and subsequent disabling of unauthorized services (such as telnet or FTP) could be deemed necessary by an organization to address temporary needs, forgetting to return the system to its original state would result in an erosion of its security profile and an increase of the overall risk of compromise.

And what about tools and utilities that could be used for harm, in the event that a system is successfully exploited? Should compilers or troubleshooting tools like trace, traceroute or Wireshark be installed on production servers, if they could end up being used to facilitate an attack? How about simply not installing unnecessary and sensitive software components in the first place?

In order to adequately protect host systems, serious consideration should be given to introduce stricter controls on system capabilities and user behavior, so that we don’t simply place the key under the doormat or inadvertently leave the windows wide-open after locking our front door.

2. Minimization

Installing only necessary operating system components is a fundamental way to eliminate risk at the source: it's impossible to use something that simply isn't there. Together with supporting policy that defines and enforces limits on what can be used and made available on production servers (and other key systems), an organization can effectively control what their environment does and how it goes about doing it, with confidence that any policy break results from an intentional act.

A seminal paper on operating system minimization from Sun Microsystems promoted the notion that the way to "reduce system vulnerabilities is to minimize the amount of software on a server" (Noordergraaf & Watson, 1999). Updates to that original paper (Noordergraaf et al., 2002) built upon the proposed methodology for meeting that objective, resulting in the development of a "cookbook" on how to reduce the software footprint of operating systems in a controlled, deliberate, systematic and reproducible fashion. Current-day initiatives, such as the Damn Small Linux (DSL) distribution and the various "Just enough Operating System" (JeOS or "juice") offerings, are a current reminder that platform minimization remains a relevant and meaningful strategy when deploying computing systems in production environments.

2.1. Justifications beyond reducing exposure

Aside from stripping away unnecessary functionalities to reduce vulnerability, minimization provides several additional benefits. In fact, not all minimization initiatives have security as their primary focus when developing their solutions (Damn Small Linux, 2009). These differing perspectives and driving forces, resulting in similar overall benefits, serve to demonstrate that much can be gained by using a basic grocery shopping metaphor – picking and choosing only the items that are needed.

2.1.1. Less patching

Monitoring security advisories and keeping systems updated with the latest updates is a tedious task for system administrators and security professionals, for consideration needs to be taken each and every time on how system functionality and availability could be impacted through installation of a software correction. The benefits

of counteracting the "vulnerability du jour" or a nagging software bug with a patch often has to be balanced against the risk that software dependencies could be disturbed, updates don't install correctly, or in-house code simply stops functioning.

A minimized platform has fewer components, and as a result, the need for patching is reduced. An operating system installation containing a quarter of the available software packages in a standard distribution could help rationalize advisory monitoring and reduce patching activities by a corresponding factor of 75%.

2.1.2. Reduced dependencies

Software dependencies tend to be difficult to manage; and as software becomes more modular and code reuse grows in popularity, the risk increases when it comes to update software that is relied upon by many other implementations. (For example, OpenSSL cryptographic libraries are often used by other software components.)

By minimizing the software profiles of systems, patching is facilitated as there are fewer interdependencies between components, resulting in a quicker response time in closing vulnerabilities.

2.1.3. Increased performance

When system owners propose hardware upgrades to upper management, they are often told to "do more with less" and encouraged to find ways to increase performance of their aging equipment without incurring additional costs. Depending on the operating system and the host system's baseline configuration, tuning options to squeeze out better performance may appear limited.

Smaller installation footprints generally result in faster boot times and quicker shutdowns, for there are fewer processes to bring up and tear down. As well, reductions in RAM and hard drive memory usage could be significant, resulting in quicker system response and shorter backup times.

2.1.4. Improved control on user behavior

As organizations grow in size, there often comes a need to standardize processes, tools and methodologies. Without standardization in the use of compilers, operating systems and programming languages, developers would have a very difficult time to pick

up where another person left off, and system owners would be left with software painfully difficult to maintain.

From an operations perspective, there is perhaps more of a challenge in standardizing ways of working, but the challenges are fundamentally the same. Three different system administrators working in the same team may each prefer the use of their own favorite Unix shell (for use during interactive login sessions). Yet all three may confront a security analyst in unison to defend and justify the need to include a suite of compilers, libraries, interpreters and troubleshooting utilities in production systems, in case of emergency.

Limiting shell installation to the Bourne shell (for administrative use) would effectively force users to use the same tool, potentially eliminating weaknesses in certain shell implementations and reducing the opportunity to develop scripts using shells that may be best suited for interactive use (such as tcsh). By categorically disallowing the installation of system tools and utilities on production systems, malicious users have greater difficulty to audit nodes, perform network reconnaissance and build attack tools, thereby hindering their progress as they plan their next course of attack.

Perhaps there is no better way to discourage the use of insecure protocols like telnet and FTP than by not having them made available. Decisions on what administrative and troubleshooting tools to include on systems requires careful consideration between potential exposure and ease of use; in more controlled environments, administrators would be forced to install such tools on an as-needed basis, then return the target nodes to their original software profiles upon task completion.

2.2. Revisiting the disabling option

One of the concerns often brought up, when presenting a platform minimization strategy, is the fear that removing certain components will hinder the ability of staff to troubleshoot and fix problems, and the uncertainty that the system will continue to function correctly if too many components are stripped away. By only disabling features and components, critics maintain that dependencies and capabilities are retained and overall risk is reduced to acceptable levels.

But what other risks are maintained by not removing unnecessary or undesirable functionalities? The concerns with regards to "crippling" troubleshooting capabilities are ultimately a business decision, and can be effectively reconciled through the establishment of some ground rules and a sound process. The risks associated with breaking functionality by reducing the operating system footprint are addressed later in this paper, through the use of a bottom-up package addition methodology (as opposed to a stripping-down process).

With regards to retaining disabled services, there is also a patching aspect to consider and a myth to debunk. Any argument stating that disabled components do not need to be patched is fundamentally flawed: if there is sufficient justification to disable components or services because they are insecure, why should vulnerable, unpatched versions of those components or services lie dormant on critical computing systems and be made available for use in an emergency? Disabled components must maintain the same current patch levels as active components, in order to mitigate the increased risk introduced whenever they are activated.

Finally, there is the human factor to consider. Once a troubleshooting activity is completed and functionality is restored on a failed system, staff stress levels drop dramatically and there is a real possibility that temporary measures don't get rolled back to pre-incident settings; after all, why touch something that started working properly again? In order to control the human factors of forgetfulness and reluctance, a robust event handling process should include detailed note taking, along with an obligation to return security-sensitive system configurations to their original approved state.

2.3. Defining the business case

A general trend followed by operating systems is continuous growth, with new releases introducing additional features and capabilities, bundled together with software corrections collected since the previous release. This bloating of platform kernels (Schneier, 2000) and footprints have progressively made operating systems more feature-rich and attractive from a marketing perspective, but it has also created havoc on the software industry: many developers insist on using the latest and greatest technologies when performing their craft, and perhaps justifiably so; yet alongside every new software

component developed in Ajax, Python or Ruby, there can be other components compiled with several different versions of gcc, and still others requiring three different versions of Java runtime environments running side-by-side for support.

At a high level, complexity can be seen as a shifting constant; just as driving skills become less essential when behind the wheel of a car equipped with power steering and ABS brakes, adding software components and facilitating code development on one side inevitably triggers additional costs in knowledge acquisition, license management, platform maintenance, system overhead, solution performance, software administration and security management.

2.3.1. Simplifying the landscape

"...it is unreasonable to expect software to not have security bugs. The simpler the software is, the fewer bugs it will have." (Schneier, 2000)

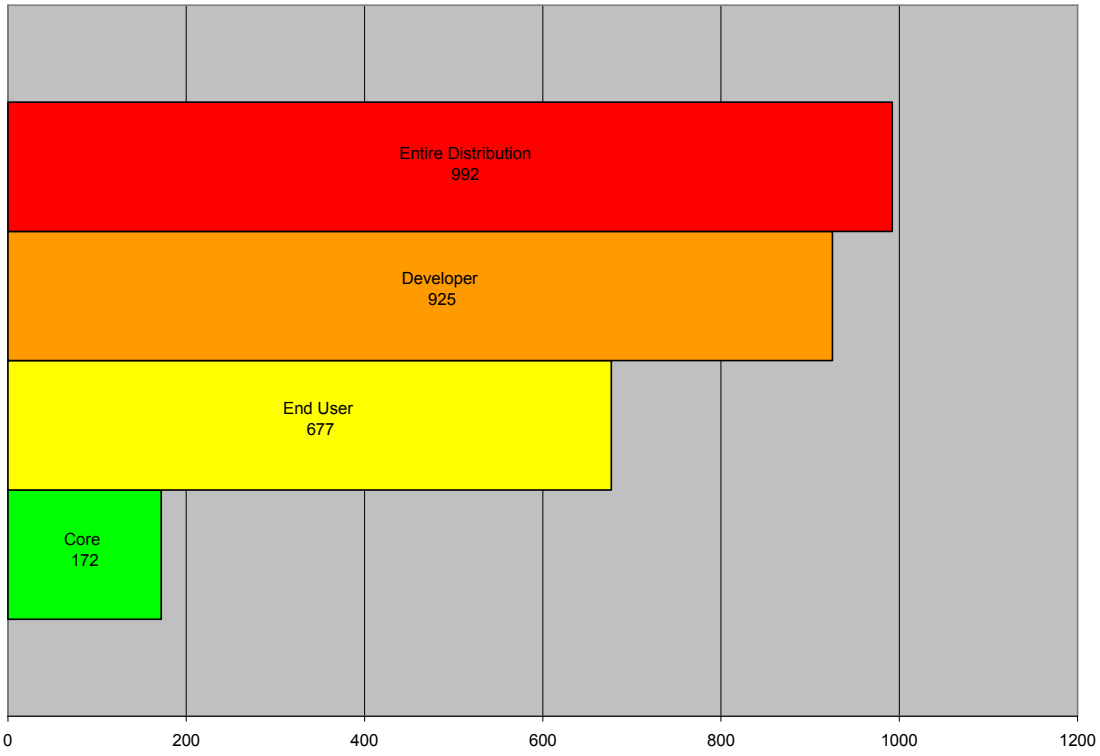
So how much platform is enough? Can one realistically reduce the footprint of an operating system to a mere fraction of a software distribution, which retaining all necessary and desired functionality? After all, when considering what to chop out, it is easy in this case to see the forest for the trees (Wuerthner, 2009), and end up justifying the inherent value of the whole.

Looking back at Noordergraaf's body of work on Solaris minimization, there is an opportunity to perform a paradigm shift and move focus from the sum of the whole towards the actual items of interest: the components that are needed to support desired functionality. In the case of that particular operating system, there were generally four different installation clusters made available to begin controlling OS footprint size: a Core cluster (the smallest package that included all necessary hardware support); an End User option (which included support for various desktops); the Developer option (that added libraries and compilers); and the Entire Distribution, also known as the OEM cluster (Noordergraaf, 2002).

"Because it is so difficult to determine the minimal set of necessary packages, system administrators commonly just install the Entire Distribution cluster. While this may be the easiest to do from the short-term perspective of getting a system up and

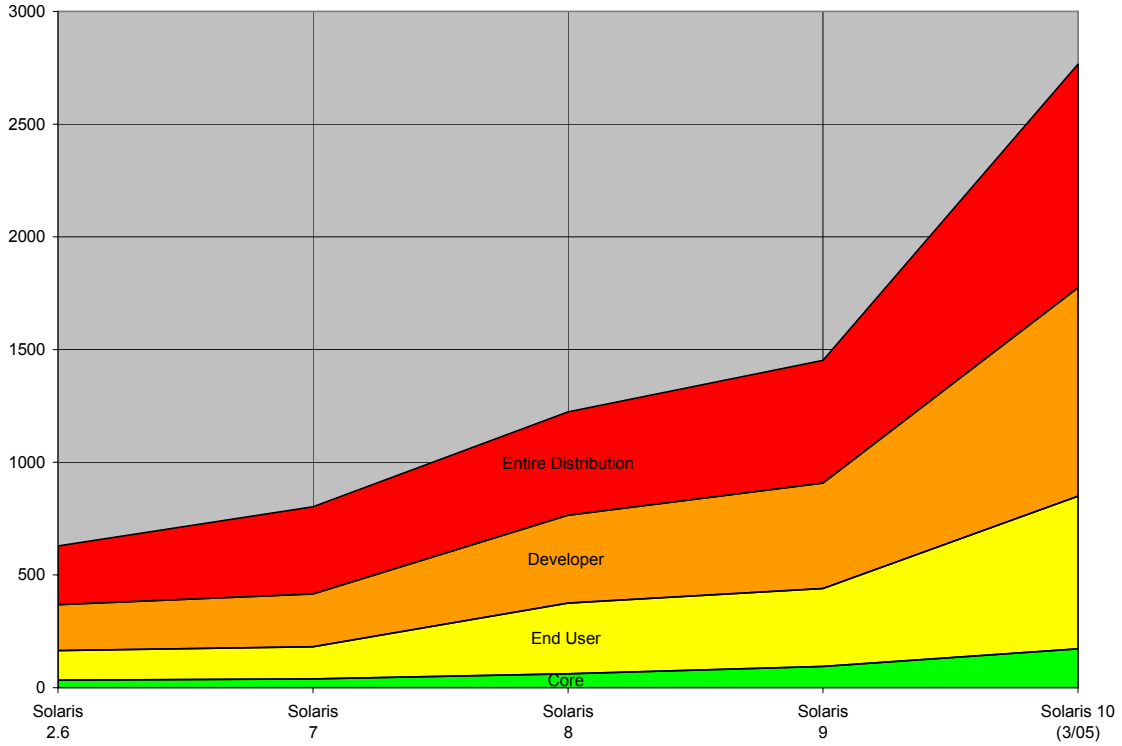
running, it makes it considerably more difficult to secure the system." (Noordergraaf and Watson, 1999)

A size comparison of the Solaris 10 (3/05) installation cluster packages (Noordergraaf, 2002) clearly shows that the vast majority of software packages included in comprehensive operating system distributions are geared towards user interfaces and software development support (package totals included under each installation cluster):



Due to ever-increasing collaborative efforts on the parts of members of the development community at-large, comprehensive Open Source operating system distributions tend to include many more components than their commercial counterparts. Growth in the number of install packages is further increased with the splintering of existing packages into smaller modular components in order to facilitate teamwork and control update impacts on an entire code base. The results in terms of numbers can be staggering: between OpenSolaris releases 2008.05 and 2009.06 (Oracle Corporation, 2010), the sum of packages ballooned from 1224 to 1709; statistics from Debian GNU/Linux are even more eye-popping, with versions 4 (etch) and 5 (lenny) containing 23,156 and 28,250 packages respectively (SPI Inc., 2010).

The growth trend of operating systems over time is real and significant, in that it affects minimization effort. The following illustration (keying on OS package numbers over successive product releases) of Solaris package growth (Noordergraaf, 2002) shows that this trend also affects each of the various target audiences of an operating system distribution:



2.3.2. Building appliances

The fundamental difficulties in performing minimization are to identify all necessary components and to capture all product dependencies. As package management utilities (such as Solaris' pkg, Red Hat's RPM and Debian's APT) can resolve many of the dependency issues automatically, installation concerns should first focus on which functions are needed to support the business logic of the target system.

Generally, production servers are deployed to perform a specific task such as serve web pages, manage databases or warehouse source code. Once the core function is identified and a software solution for this function has been selected, it is a relatively simple exercise to start putting together a list of third-party software needed to support

the business solution: developers of an in-house software solution should be able to quickly identify what extra code they acquired to test their prototypes; both commercial and Open Source third-party software tend to publish listings of other necessary third-party software required for functional support.

Where this declarative model breaks down is when it comes to identifying which individual packages are required of host operating systems, as such dependencies continue to be rarely defined in software application documentation (Noordergraaf, 2002). This is because minimization is considered to be a time-consuming task, and because explicit support of a range of platforms (including a variety of Linux distributions) would require package identification by the supplier for each single platform variant. Regrettably, the burden of this responsibility falls on the motivated individual – the prudent system administrator or the demanding security specialist.

At the end of the day, the intent is to produce an appliance: a system assembled to perform a dedicated task, such as a toaster or a network router, tuned in such as a way as to perform this task in an efficient and effective manner. As is the case in manufacturing environments, the objective is to put together the best product at the right price, using only the components needed in order to cut downstream costs to a minimum, and limit the amount of after-sales customer support to acceptable levels.

The preferred way to begin selecting operating system components is by adopting a core installation package cluster, unless the OS supplier has a predefined package profile that corresponds to the server's core function; then going through that initial package listing and identifying what is and what isn't needed from a hardware and software perspective. It is much easier from a testing perspective to add components to introduce functions than to take a fully operational system, drop a few components, hope for the best, and run a full battery of tests to see if all business functions are still operational. When sifting through package listings, it is recommended to retain only those components that are found to be definitely required, rather than err on the side of caution; the minimization methodology defined later in this paper has been designed to identify and capture any missing components along the way.

The resulting listing would then become the starting point for the minimized

operating system platform.

2.3.3. Mapping dependencies

The mapping of dependencies is the second big part of the minimization effort. As alluded to earlier, much of this effort is alleviated with the prudent use of the package management system in place. Package management systems track interdependencies and are programmed to maintain relationships, and although errors can be made through data entry when defining these relationships, any benefit of breaking these bonds during system installation – in hopes of eliminating a few extra packages – would only introduce a loss of reliability in the integrity of the database. It is strongly recommended to bring down the number of OS packages to the minimum level possible, all while maintaining pre-defined package dependencies.

A useful strategy for mapping dependencies and justifying the inclusion of each OS package is to log the intra-relational mapping and record the underlying reasoning, either in a text document or a spreadsheet. Mapping dependencies may result in voluminous documentation, but it is a valuable exercise in learning how the target system functions from an outside view (as opposed to a programmer's inside view).

2.4. Preparing for deployment

Once there is a buy-in and a commitment for minimizing OS platforms, and after the preliminary identification of software components has been completed, it's time to put the rubber on the road and bring the technical implementation into motion. On the other hand, for those motivated individuals unable to secure buy-in for this type of activity and somehow find the bandwidth to invest in better securing their systems, there is hope: minimization can be effectively performed as an intermittent background activity using virtual machines. With respect to investing time and effort in minimization, sometimes "it's easier to get forgiveness than permission" (McKenzie, 2009).

3. Implementation

There is currently a renewed interest in minimization, witnessed by the increasing number of JeOS projects publishing compact operating systems distributions for download: some launched as niche projects by those looking for an improved platform to fit their needs (RKO Security, 2009); others directly promoted by OS suppliers in response to that market need (Canonical Inc., 2010). In both cases, the preferred direction is towards the development of virtual appliances – an operating system and software solution installed inside a virtual server container (Novell Inc., 2008).

An audit of recently published security-related Linux books demonstrates that the deployment of OS profiles tuned to production needs is generally recommended, sometimes because particular distributions do not offer full install or minimal install options (Negus & Foster-Johnson, 2009). But only in rare cases (Rankin & Hill, 2009) is any detail provided on how to build popular server types, regrettably with little mention of methodologies on whittling down packages to the minimum from a cluster baseline. It is for this reason that Noordergraaf's body of work in this area will once again be visited, before presenting an updated methodology optimized to take advantage of improvements in today's most popular package management systems.

3.1. Methodology

The Solaris minimization methodology called for an installation of a core package cluster and the addition of required supporting software, followed by patching, subsequent removal of unnecessary packages and OS configuration (Noordergraaf, 2000). Due to the robustness of today's installation tools, the mature yet fragile nature of popular package management systems such as RPM and APT, as well as the inherent responsiveness of the Open Source community to correct faults found in software in a prompt manner, the following order for minimizing UNIX-based platforms is proposed:

3.1.1. Installation of a minimal subset of packages

Modern installation programs, such as the Red Hat (Graphical or Text Mode) Installation Program User Interfaces, provide the ability to not only select installation

clusters (defined by Red Hat as Package Groups), but also grant the option of fine-tuning the selection of OS packages during the installation setup. The installation programs maintain package relationships, and will install any additional support packages that were not explicitly selected during the fine-tuning process.

Once installation is completed, the system should be shutdown and rebooted, all while monitoring the console messages for any error messages. (Monitored shutdown and reboots are also suggested after each remaining activity.)

By adding desired components (such as SSH and NTP support) during this initial selection process (which may not necessarily come with core clusters), and removing components from the core that are deemed unnecessary, the amount of post-installation work can be measurably reduced.

3.1.2. Parallel installation of an OS reference system

In order to facilitate the tracking and tracing of software dependencies, it is strongly recommended to maintain a reference system in parallel, one that includes a full software installation of the adopted operating system. By reproducing installations in both minimized and full OS environments, one should be able to ascertain quickly if any installation and functional failures are due to missing infrastructure or to other factors.

While the temporary use of a second system may have been considered prohibitively expensive in the past due to doubled hardware needs, the availability and ease-of-use of today's virtual machines have eliminated that financial consideration, even if the target system is planned for a bare metal installation on new server hardware.

3.1.3. Removal of unnecessary packages

Once installation in completed, the package management system is used to consult what has been deployed on the target system and further reduce the footprint of the operating system. As was the case during initial installation, the package management system will alert the installer of any problems with respect to maintaining intra-package relationships.

But wasn't the selection of packages already performed before installation? Additional packages sometimes do get introduced during an initial installation, over and

above those required for supporting selected packages, quite possibly to maintain features desired by the OS supplier.

3.1.4. Configuration and hardening of the platform

Much of the basic OS configuration – such as regional settings, networking and disk partitioning – tends to be defined during the installation process. Hardening of the platform should be performed as early as possible after initial installation so as to reduce the exposure of the newly installed system.

The Center for Internet Security's benchmarks and audit tools can help guide hardening activities: benchmarks are free configuration guides made available for a broad range of popular UNIX, Linux and Windows operating systems; most audit tools require CIS membership, and are used to perform automated host-level benchmark assessments (CIS, 2010).

3.1.5. Installation of relevant patches

Depending on the installation method, patching may have already been performed. Certainly, using a live Internet-based installation should result in the deployment of a server that could only require a few extra corrections to be up-to-date, but some organizations implement code controls and require the use of only pre-approved software versions. This would preclude the use of this type of installation method.

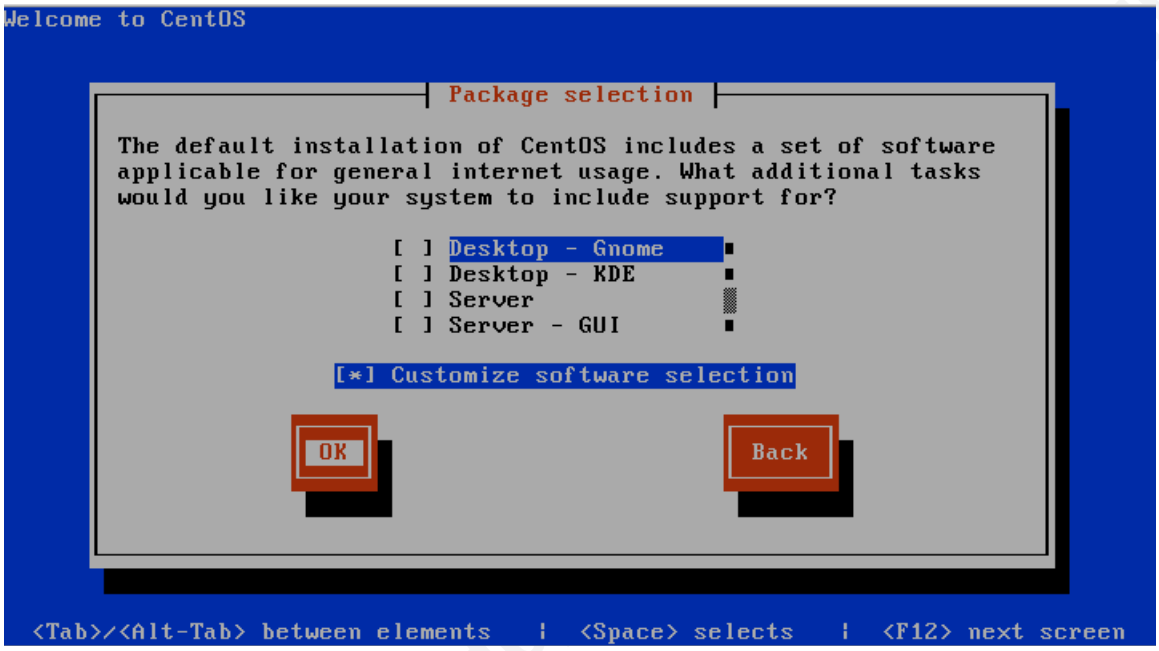
Alternately, the deployment of patches could be performed immediately after initial installation of the OS, followed by package removal, but is not recommended because of the risks involved with breaking package relationships by introducing new components with old at an earlier point in the process. Also, by holding off on patching until the last of these five steps, the target system is deployed in a controlled environment and only exposed to (local or remote) update services after being hardened.

3.2. Example

In deference to Noordergraaf's contributions to the study of minimizing UNIX-based operating systems, the following section briefly highlights how package selection and subsequent removal can be performed, through a sample deployment of an Apache web server atop a CentOS implementation, residing upon a VMware virtual machine.

3.2.1. Selecting clusters and packages before installation

During the initial phases of the CentOS installation process, the installation program opens a series of windows to solicit inputs for language and keyboard options, disk partitioning information, network and hostname configuration, plus time zone and root password choices, before presenting the package selection screen:



On a CentOS 5.2 text based installation, a "package selection" of only the "Server" profile resulted in the installation of 450 operating system packages.

By entering the "Customize software selection" sub-menu, one can pick and choose the Package Groups that ended up being tagged through the profiles chosen in the previous screen. In this case with only the Server option selected, fourteen Package Groups were highlighted for installation; by dropping 11 groups and retaining only Base, Editors and Systems Tools, the number of installed packages went down to 391.

While this resulted in a significant 13% reduction in packages, that figure did not remotely correspond to the 79% reduction in the number of Package Groups. A subsequent installation attempt with no profile selection and no Package Groups selected generated a CentOS 5.2 system installation comprised of 150 packages (see Appendix A), proof that there is some type of logic inside some installation tools to ensure that installed systems boot and provide a predefined-as-critical set of processes and services, even after aggressive platform minimization.

3.2.2. Removing packages after installation

A glance at a corresponding (same CentOS/Red Hat 2.6.18-92 kernel) Orange JeOS VMware image (RKO Security LLC, 2008) showed a package listing of 185:

```
Orange JeOS Core release 1.8.6-69
Kernel 2.6.18-92.1.13.el5 on an i686
appliance login: root
Password:
Last login: Sun Jan 24 21:44:40 on tty1
[root@appliance ~]#
[root@appliance ~]# rpm -qa | wc -l
185
[root@appliance ~]# _
```

A comparison between the two minimized environments showed that there were some small discrepancies that could be used to reduce package numbers further on either side: the base profile developed from the proposed methodology included some IPv6 support (dhcpv6_client) and an OpenSSH client (openssh-clients) that could be dropped; while the Orange JeOS platform had some superfluous components of it's own such as a vector graphics library (cairo), a font configuration utility (fontconfig) and some icon support (hicolor-icon-theme) that also has no real place on production servers that don't need graphical support.

Interestingly, some common components in both platforms included some packages that appear to be excellent candidates for removal, which somehow made it through the restrictive selection process – including a legacy text editor (ed) and a wireless networking configuration utility (wireless-tools). Removal attempts on both of these components should that some additional investigations may have to be performed:

```
[root@localhost ~]# rpm -e ed
[root@localhost ~]# rpm -e wireless-tools
error: Failed dependencies:
        libiw.so.28 is needed by (installed) rhpl-0.194.1-1.i386
[root@localhost ~]# _
```

In the example above, a library used by some CentOS programs (rhpl) has a dependency on a file (libiw.so.28) that is part of the wireless-tools package. If that library is not required by any other installed packages, then "rhpl" and "wireless-tools" could be removed in succession:

```
[root@localhost ~]# rpm -e ed
[root@localhost ~]# rpm -e wireless-tools
error: Failed dependencies:
        libiw.so.28 is needed by (installed) rhpl-0.194.1-1.i386
[root@localhost ~]# rpm -e rhpl
[root@localhost ~]# rpm -e wireless-tools
[root@localhost ~]# _
```

Other package removal scenarios may not be so simple to resolve. Attempts to remove certain packages that may appear to be superfluous at first glance, such as support for the Python programming language, may show dependencies on components that are considered mission critical – such as "yum", a popular RPM package management and package update utility. By taking the time to map the dependencies in document or spreadsheet form, and shaving off any remaining unnecessary OS packages, the informed analyst will eventually be able to come up with a final "gold image" baseline that represents the smallest core operating system available for his environment.

The "rpm –e" command can be run with a space-separated listing package names in cases where there are cross-dependencies between several packages targeted for removal.

Package removal is an iterative process. If case of need, packages can be simply reinstalled with the same RPM package management command. Once the fine-tuning is completed, the final output of this exercise is a minimized platform that could be used as a "gold image" baseline OS system for many future server deployments.

3.2.3. Adding packages after installation

Once the minimized environment is defined, the next step is to introduce the business logic that defines the system in question. In this particular example, the target system will be a web server running the latest stable version of Apache HTTP Server. In order to be able to quickly introduce security patches to such a sensitive piece of software, and not have to rely on an intermediary such as an OS supplier that bundles the same component, the application will be sourced directly from the Apache Software Foundation's HTTP Server web site (http://httpd.apache.org).

An attempt to install the component on a minimized CentOS platform flagged several failed dependencies; in this particular case, four necessary libraries were missing:

```
[root@zarf ~]# rpm -i httpd-2.2.3-1.i386.rpm
warning: httpd-2.2.3-1.i386.rpm: Header V3 DSA signature: NOKEY, key ID 751d7f27
error: Failed dependencies:
        libcrypto.so.4 is needed by httpd-2.2.3-1.i386
        liblber-2.2.so.7 is needed by httpd-2.2.3-1.i386
        libldap-2.2.so.7 is needed by httpd-2.2.3-1.i386
        libpq.so.3 is needed by httpd-2.2.3-1.i386
        libssl.so.4 is needed by httpd-2.2.3-1.i386
[root@zarf ~]# _
```

By consulting a reference system running a full install of the operating system, one can find out if each of the necessary library files can be sourced from the OS distribution. On the full CentOS 5.2 install, a quick search of the string "libssl.so" on a filesystem listing discovers a soft link for Version 6 of the file, so that particular dependency can be satisfied through the installation of its' parent package. Confirming the location of that link with the "find / -name libssl.so.6" command, and subsequently using that result as a parameter to the "rpm –qf" command, one can identify the missing OS package that would have to be added to the platform.

```
[root@CentOSfull ~]# rpm -qf /lib/libssl.so.6
openssl-0.9.8b-10.el5
[root@CentOSfull ~]# _
```

Package addition is also an iterative process. By persistently tracking all package dependencies, configuration management is performed at the server level through the justification of every single component deployed on production systems.

4. Conclusion

Platform minimization, implemented through careful selection and installation of only the OS packages required for supporting necessary system functions, is a fundamentally effective method for hardening computing systems. By limiting the at-hand availability of system services, utilities and support functions to only those needed for desired system and user behavior, the number of potential vulnerabilities is measurably reduced. Additional benefits associated with minimization include a reduced need for patching, facilitated software management due to fewer component intra-dependencies, and increased system performance as a result of lower overhead.

Yet despite a renewed interest in recent years, platform minimization remains a tough sell. This is due in some degree to the reluctance of some OS suppliers to embrace the benefits of mitigating risk by limiting deployed system footprints, because of the challenges associated with supporting wide ranges of implementations of their products:

"The majority of software vendors (including Sun) do the majority of their testing using systems that have been installed using the complete set of Solaris OS software (...). Testing is rarely completed using reduced or minimal configurations." (Sun Microsystems, 2006).

Minimization is also considered to be grunt work, as it requires functional analysis, detailed record keeping and a disciplined multi-cycle iterative approach. In this day and age where security professionals are often judged in terms of their ability to perform "ethical hacking" by successfully executing penetration tests, the rewards of deploying robust solutions tuned to business needs may have to be measured in terms of reduced maintenance efforts instead of peer recognition.

5. Appendix A: CentOS 5.2 Base Packages

audit-libs
audit-libs-python
authconfig
basesystem
bash
beecrypt
bzip2-libs
centos-release
centos-release-notes
checkpolicy
chkconfig
coreutils
cpio
cracklib
cracklib-dicts
cryptsetup-luks
cyrus-sasl-lib
db4
dbus
dbus-glib
device-mapper
device-mapper-event
device-mapper-multipath
dhclient
dhcpv6-client
diffutils
dmidecode
dmraid
e2fsprogs
e2fsprogs-libs
ecryptfs-utils
ed
elfutils-libelf
ethtool
expat
file
filesystem
findutils
gawk
gdbm
glib2
glibc
glibc-common
gnu-efi
grep
grub
gzip
hal
hdparm
hwdata
info
initscripts
iproute
iptables
iptables-ipv6
iputils
kbd
kernel
keyutils-libs
kpartx
krb5-libs
kudzu
less
libacl
libattr
libcap
libgcc
libgcrypt
libgpg-error
libhugetlbfs
libselinux
libselinux-python
libsemanage
libsepol
libstdc++
libsysfs
libtermcap
libusb
libuser
libvolume_id
libxml2
libxml2-python
lvm2
m2crypto
MAKEDEV
mcstrans
mingetty
mkinitrd
mktemp
module-init-tools
nash
ncurses
net-tools
newt
nspr
nss
openldap
openssh
openssh-clients
openssh-server
openssl
pam
passwd
pciutils
pcre
pm-utils
policycoreutils
popt
prelink
procps
psmisc
python
python-elementtree
python-iniparse
python-sqlite
python-urlgrabber
readline
redhat-logos
rhpl
rootfiles
rpm
rpm-libs
rpm-python
sed
selinux-policy
selinux-policy-targeted
setools
setserial
setup
shadow-utils
slang
sqlite
sysfsutils
sysklogd
system-config-securitylevel-tui
SysVinit
tar
tcl
tcp_wrappers
termcap
tzdata
udev
udftools
usermode
util-linux
vim-minimal
wireless-tools
yum
yum-metadata-parser
zlib

6. References

Avolio, F. (2009). Firewalls and Internet security, the second hundred (Internet) years. *The Internet Protocol Journal*, 12(4), Retrieved from http://www.cisco.com/web/about/ac123/ac147/ac174/ac200/about_cisco_ipj_archive_article09186a00800c85ae.html

Brining, S. (2008, August 26). *When a firewall is not enough*. Retrieved from http://www.imakenews.com/hosting/e_article001180714.cfm?x=b11,0,w

Canonical Inc. (2010). *JeOS: Ubuntu Server Edition*. Retrieved from http://www.ubuntu.com/products/whatisubuntu/serveredition/jeos

CIS. (2010). The center for Internet security. Retrieved from http://cisecurity.org

Damn Small Linux: biz-card desktop OS. (2009). Retrieved from http://www.damnsmalllinux.org/

Leiber, J., & Stoller, M. (1968). Is that all there is? [Recorded by Peggy Lee]. On *Is that all there is?* [Medium of recording: Record] Los Angeles: Capitol. (1969) *Hat Enterprise Linux bible*. Indianapolis, IN: Wiley Publishing, Inc.

Noordergraaf, A., (2000). *Solaris operating environment minimization for security: a simple, reproducible and secure application installation methodology: updated for Solaris 8 Operating Environment*. Palo Alto, CA: Sun Microsystems, Inc. Retrieved from http://www.sun.com/blueprints/1100/minimize-updt1.pdf

Noordergraaf, A., (2002). *Minimizing the Solaris operating environment for security: Updated for Solaris 9 Operating Environment*. Palo Alto, CA: Sun Microsystems, Inc. Retrieved from http://www.sun.com/blueprints/1102/816-5241.pdf

Noordergraaf, A., et al. (2002). *Enterprise security: Solaris operating environment*. Santa Clara, CA: Sun Microsystems Press.

Noordergraaf, A., & Watson, K. (1999). *Solaris operating environment minimization for security: a simple, reproducible and secure application installation methodology*. Palo Alto, CA: Sun Microsystems, Inc. Retrieved from http://www.sun.com/blueprints/1299/minimization.pdf

Northcutt, S. (2007, February 26). *The uniform method of protection to achieve defense-in-depth*. Retrieved from http://www.sans.edu/resources/securitylab/367.php

Novell Inc. (2008, August). *Technical white paper: advantages of building virtual appliances on SUSE Linux Enterprise Server* [462-002088-001]. (Adobe PDF version), Retrieved from

http://www.novell.com/rc/docrepository/public/37/basedocument.2008-08-12.5466048836/Advantages_of_Building_Virtual_Appliances_on_SUSE_Linux_Enterprise_Server_Technical_White_Paper_en.pdf

Oracle Corporation. (2010). *OpenSolaris: package catalog*. Retrieved from http://pkg.opensolaris.org/release/en/catalog.shtml

Rankin, K., & Hill, B.M. (2009). *The official Ubuntu server book*. Boston, MA: Pearson Education.

Red Hat Inc. (2005). *Red Hat Enterprise Linux 4: installation guide for x86, Itanium, AMD64, and Intel extended memory 64 technology (Intel® M64T)* [rhel-ig-x8664(EN)-4-Print-RHI (2004-09-24T13:10)]. (Adobe PDF version), Retrieved from http://www.linux-books.us/red_hat_enterprise_0001.php

RKO Security LLC. (2008). *Orange JeOS downloads*. Retrieved from http://www.rkosecurity.com/oj_download.html

RKO Security. (2009, January 20). *Orange JeOS: small & good for you*. Retrieved from http://orangejeos.sourceforge.net/

Schneier, B. (2000). *Secrets and lies: digital security in a networked world*. New York, NY: John Wiley and Sons, Inc.

Shingledecker, R., Andrews, J., & Negus, C. (2007). *The official Damn Small Linux book: the tiny adaptable Linux that runs on anything*. Upper Saddle River, NJ: Prentice Hall PTR.

SPI Inc., (2010). *Debian: packages*. Retrieved from http://www.debian.org/distrib/packages

Sun Microsystems. (2006). *Rules of engagement for the support of reduced or minimal configurations*. Palo Alto, CA: Sun Microsystems, Inc. Retrieved from http://www.opensolaris.org/os/community/security/files/minimization-support-rules-ext.pdf

Sun Microsystems Security Engineers. (2009). *Solaris 10 security essentials*. Boston, MA: Prentice Hall PTR.

von Hagen, W. (2001). *Installing Red Hat Linux 7*. Indianapolis, IN: SAMS Publishing.

Wuerthner, G. (2009, March 30). *See the forest for the trees*. Retrieved from http://www.newwest.net/topic/article/see_the_forest_for_the_trees/C564/L564/

Last Updated: September 18th, 2020

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Australia Spring 2020	**, AU**	**Sep 21, 2020 - Oct 03, 2020**	**Live Event**
SANS FOR500 Milan 2020 (In Italian)	**Milan, IT**	**Oct 05, 2020 - Oct 10, 2020**	**Live Event**
SANS October Singapore 2020	**Singapore, SG**	**Oct 12, 2020 - Oct 24, 2020**	**Live Event**
SANS SEC504 Rennes 2020 (In French)	**Rennes, FR**	**Oct 19, 2020 - Oct 24, 2020**	**Live Event**
SANS SEC560 Lille 2020 (In French)	**Lille, FR**	**Oct 26, 2020 - Oct 31, 2020**	**Live Event**
SANS Tel Aviv November 2020	**Tel Aviv, IL**	**Nov 01, 2020 - Nov 06, 2020**	**Live Event**
SANS Sydney 2020	**Sydney, AU**	**Nov 02, 2020 - Nov 14, 2020**	**Live Event**
SANS Secure Thailand	**Bangkok, TH**	**Nov 09, 2020 - Nov 14, 2020**	**Live Event**
APAC ICS Summit & Training 2020	**Singapore, SG**	**Nov 13, 2020 - Nov 21, 2020**	**Live Event**
SANS FOR508 Rome 2020 (in Italian)	**Rome, IT**	**Nov 16, 2020 - Nov 21, 2020**	**Live Event**
SANS Oslo Local November 2020	**Oslo, NO**	**Nov 23, 2020 - Nov 28, 2020**	**Live Event**
SANS Wellington 2020	**Wellington, NZ**	**Nov 30, 2020 - Dec 12, 2020**	**Live Event**
SANS OnDemand	**OnlineUS**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**