An Automated Binary Security Update System for FreeBSD

Colin Percival
Computing Lab, Oxford University
colin.percival@comlab.ox.ac.uk

Abstract

With the present trend towards increased reliance upon computer systems, the provision and prompt application of security patches is becoming vital. Developers of all operating systems must generally be applauded for their success in this area; systems administrators, however, are often found lacking.

Anecdotal evidence suggests that for FreeBSD much of the difficulty arises out of the need to recompile from the source code after applying security patches. Many people, after spending years using closed-source point-and-click operating systems, find the concept of recompiling software to be entirely foreign, and even veteran users of open source software are often less than prompt about applying updates. Providing these people with a binary option should significantly improve the rate at which security updates are applied.

This paper describes an automated system for building and distributing binary security updates for FreeBSD, and describes the challenges encountered. I also describe some of the limitations of this system, and discuss some possibilities for future work.

1 Introduction

Over the past few years, there has been a trend towards a much more rapid exploitation of security holes. It has been shown with honeypots that insecure systems are often compromised within days or hours of being connected to the internet [An02, Ho02]; it has even been suggested that a significant proportion of systems connected to the internet could be compromised by a sophisticated worm within 30 seconds [SPW02].

At the same time, there is a constant influx of new users into the FreeBSD community; even with detailed instructions, traffic on the FreeBSD mailing lists indicates that a large number of people find the task of applying security patches and rebuilding affected programs to be difficult and/or confusing. Given that releases are on average several months – and several security holes – old by the time they are installed, the possibility arises that a new user will find his system compromised before he has a chance to bring it up to date.

Furthermore, there are some circumstances where building from source is undesirable. Some embedded systems might lack sufficient disk space to store the entire source and object trees; some system administrators remove part or all of the build toolchain in an (arguably misguided) attempt to thwart any attempt to build a rootkit; and the purveyors of application-specific 'toasters' might very likely wish to keep the complexity of building from source entirely hidden from their users.

For these reasons, we believe that the provision of a system of binary security updates is absolutely critical.

2 Previous work

A large number of binary update systems have been created for various applications and operating systems, for both security updates and more general software updates. We first consider systems specific to security updates.

Between June 2001 and May 2002, many FreeBSD security advisories were accompanied by 'experimental binary upgrade' packages [SA02]. These were built by hand based on (human) consideration of which binaries should have been modified by a given source patch, and distributed in the standard FreeBSD package format. When a large number of binaries were affected (for example, if a library was modified) binary upgrade packages were not provided.

A similarly experimental, but rather more limited, system has been created for OpenBSD [Ga03]. Here, portions of the 'world' are rebuilt according to instructions accompanying the official source patches, and a (hand-picked) subset of the files built are packaged into a compressed 'tar' archive, which is installed simply by extracting the new files over the old. Again, it does not appear that any attempt was made to handle patches affecting large numbers of binaries spread across the 'world'.

A more sophisticated approach was taken by a commercial service which currently provides binary updates for NetBSD [PST03]. Based on the MD5 [Ri92] digests of binaries pre- and post-patching, a list of potential distribuends is compiled. This list is then inspected by hand to remove files which are "modified but not related"; we will describe later how some binaries end up being modified even without any changes in the source tree. This hand-pruned set of binaries is then packaged into a shell script which provides the options of installing the new binaries, reverting to the previous binaries, et cetera.

Because these systems are specific to security updates, they all attempt to minimize the number of files updated, and they all include human participation in this effort. This raises a significant danger of error; even under the best of conditions, humans make mistakes, and the task of determining which files out of a given list had been affected by a given source code patch requires detailed knowledge of how the files are built. A good example of this is the SunRPC XDR library bug from March 2003 [CE03] – few, if any, people would have expected to find vulnerable XDR code in `/bin/mv` or `/bin/rm`. We argue therefore that building updates automatically has an advantage of correctness as well as an advantage of economy.

On the side of general binary updates, the field is more varied. Perhaps the best known of these is Microsoft's Windows Update, which distributes security updates, service packs, driver updates, and new versions of Microsoft 'middleware'; these are installed in the same manner as application software. Some application packaging tools also provide binary patch mechanisms; for example, InstallShield has an update service [IS03] which, depending upon the tool purchased, can replace an application entirely, distribute only modified files, or distribute only patches to the modified files.

The RedHat and Debian distributions of Linux both have binary update systems, up2date and apt-get respectively. Similar to these are portupgrade (which, as the name indicates, only upgrades software from the ports tree), and the FreeBSD binup project [Bi02], which aims to provide a general mechanism for all binary updates, but has unfortunately stalled due to a lack of developer time. All these tools work on the same general principle – everything is 'packagized', and the updating process consists simply of removing the old package and installing a new package.

3 Automated update building

In order to build binary updates without human intervention, we start with a very simple approach: Build the 'RELEASE' world in one directory, build the world based on the latest security patches in another directory, and compare. Any files which need to be included in the published update will have changed. Unfortunately, as noted earlier, the converse is not true: Some files will change every time they are built, even if they are built from the same source files; in FreeBSD 4.7, there are 160 such files, of which 128 are library archives.

Carefully examining the regions where these files differ shows the cause: They contain human-readable time and date stamps (hereafter we refer to these, along with user and host stamps, as 'build stamps'). Some of these are well known and serve obvious purposes: The kernel and boot loader, for example, display at startup the user, hostname, date, and time when they were built, and the library archives need to record timestamps so that their constituent object files can be accurately recreated; but other executables, such as those associated with perl, NTP, PPP, and ISDN, have build stamps without any apparent purpose.

In order to eliminate false positives introduced by these build stamps, we change our process as follows: We start by building the 'RELEASE' world twice, adjusting the clock to ensure that the date is different (some files contain the date, but not the time they were built), and then compare these two worlds in order to locate the build stamps. For binary files, we consider a build stamp to consist of a byte which differs between the two versions of the file and up to 128 'string' characters in either direction; for text files, we consider a build stamp to be a complete line which differs between the two versions of the file. Once we have located the build stamps, we build the new world and compare it to the release, excluding the regions previously marked as build stamps; any variation outside of those regions indicates that the relevant file needs to be distributed as part of a binary update. Finally, we rebuild the new world again and locate the new build stamps (this final step is necessary because any change to a binary is likely to move the build stamps.)

4 A few more complications

While the above procedure works for almost all files, a few need special treatment – usually in the form of cosmetic patches to the source tree. For some reason, fortune data files are randomized during the build process, even though fortune(6) already selects a fortune randomly. This causes the fortune data files to build differently every time; removing the randomization from the build process eliminates the variability without any noticeable effect.

On a related note, the compiler used for FreeBSD 4.x (gcc 2.95), in the rare case where it cannot find a programmer written global name in a given file, introduces a random string for this purpose. (In the FreeBSD 4.7 world, this only occurs when compiling the libobjc library.) Changing this behaviour to instead generate a global name by hashing the current path and the input filename removes the variability without affecting other functionality. [Si03]

When security patches are made to FreeBSD, it is standard practice to update a version string contained in the kernel. This has the advantage of making it apparent that the changes have been made; but it has the side-effect of causing the kernel to change when userland-only security fixes are applied. We override these changes.

Some of the documentation for groff uses the current date in examples; this would be handled properly as a timestamp, except that "March" is shorter than "February", and causes cascading differences in the line breaks. Modifying the examples avoids this problem.

Finally, some files are not entirely replaced during the build process: The directory used by info(1) and perl's perllocal.pod both have entries appended to them during the build process, but are never cleaned; the kernel building code keeps a count of how many times the kernel has been compiled; and some files (the kernel, modules, boot loader, and init) are backed up. Removing the info directory, perllocal.pod, the kernel compile counter, and the backup files eliminates the spurious variability which they introduce.

One additional complication is introduced by cryptographic export laws. Some files exist in multiple versions: Non-cryptographic, cryptographic, kerberos 4, and kerberos 5. We handle this by building the afflicted files, in all applicable forms, in separate directories.

Out of the patches necessary to work around these complications, only the one relating to fortune files has been incorporated into the main FreeBSD tree. Gcc and groff are 'contributed' code, and consequently local modifications are discouraged (we note, however, that the issue with gcc is likely to be corrected in a future version); and the question of kernel labelling resulted in a very lengthy debate when the current practice was first adopted and it seems unlikely to change now.

5 Distribution

Based on our generated list of which files need to be distributed, we generate an update index containing lines of the form

```
/path/to/file$oldhash$newhash
```

where `/path/to/file` is the full path to the file being updated, `oldhash` is the MD5 hash [Ri92] of the old version being replaced, and `newhash` is the MD5 hash of the new version being installed. Note that updating one file could result in several associated lines, one for each 'old version'; to handle this, we keep a list of all 'valid' old hashes by starting with the hash values from the published binary release and adding the hashes of any new files we distribute.

Along with the update index is distributed a 2048 bit public RSA key, the MD5 hash of the update index signed with the private part of the RSA key, the new versions of the files, identified by their MD5 hashes, and binary diffs generated with BSDiff [Pe03], identified by the MD5 hashes of the old and new versions. Note that these files, once created, are entirely static.

Given that the update index contains file hashes and the update index is signed, the only step which needs to be performed securely is the publication of the public key. This is done by verifying the MD5 hash of the public key; at present, a configuration file is distributed with the client software which includes the hash of a key belonging to the author (the mechanics of securely distributing application software is a bootstrapping issue and outside the scope of this paper); anyone else using this code to publish their own binary updates would naturally have to distribute their own key.

Everything else can be done insecurely: The update files can be distributed over insecure HTTP, can be mirrored easily, and can be transported via sneakernet to update a system which has no internet connection at all. This also has the advantage of allowing updates to be built on a system which is physically disconnected from the outside world, with source patches carried in and published updates carried out manually.

6 Installation

Machines attempting to update themselves first download the RSA public key and verify that it has the correct MD5 hash; the update index is then downloaded, and the signature is verified. For each line in the update index, the MD5 hash of the currently installed file is then computed; if it matches the `oldhash` value contained in the update index, the binary diff is downloaded and the new version of the file is generated; as a backup method, if the file generated from the binary diff does not have the correct hash, the entire new file is downloaded (and verified to have the correct hash).

This use of binary diffs provides a remarkable reduction in bandwidth usage. Updating a typical installation of FreeBSD 4.7 (specifically, one which includes cryptography, but does not include either version of Kerberos) to include all the applicable security fixes as of mid-June 2003 involves replacing 97 files which total 36MB. The binary diffs for these total 621kB, a reduction by a factor of 58. Even if all the HTTP/TCP/IP overhead is added, the total bandwidth required for updating such a system is under 1.6MB – less than half of the 3.6MB used by cvsup [Po02] when performing the same update on the source tree. Indeed, based on an estimate from Netcraft that there are between 50 and 60 thousand publicly accessible web servers running FreeBSD worldwide [Pr03], and data from FreshPorts [La03] which suggests that web servers constitute slightly less than half of the machines running FreeBSD around the world, we believe that it would be possible for a single low-end server to provide binary updates to every FreeBSD system in the world within a single day.

After the updated files have been fetched and/or generated via patches, the updates are installed by backing up the old files and moving the new files over the old (subject to maintaining permissions, ownership, and file flags). Any supplemental tasks necessary for the updates to take effect (restarting daemons, recompiling staticly linked application software which uses modified libraries, and/or rebooting) is left to the system administrator.

It is important to note that this process is entirely stateless; no database is kept of which updates have been installed – instead, at each point, the currently installed files are examined to determine if they are ‘old’. There is no mechanism for installing some, but not all, available updates – we assume that nobody would wish to patch some, but not all, security holes; indeed, there is no concept of an individual ‘update’. Since the updates are produced by comparing the results of builds at various points along a security branch, there is no mechanism for identifying which particular security advisory corresponds to particular binary changes (unless, of course, there has only been one advisory in the applicable time window). Any administrator wishing to verify that he has not forgotten to update a system must simply run the client software; indeed, we encourage all potential users (i.e. people who started from a binary install and have not recompiled any part of the world) to set a cron job to run the update client.

7 Caveats

There are a few problems with the approach we take. First, because we rely upon the MD5 hash of currently installed files to determine which files need to be updated (and, in the case of export differences, which new version should be installed), any variation in the installed files will result in updates not being performed. This means that the set of potential users is restricted to those who have performed a binary install and not recompiled any FreeBSD files. We do not consider this to be a serious limitation, considering that our stated target audience is those users who are unable or unwilling to recompile from source.

Another limitation arises from the fact that we only *replace* files, rather than adding or removing them. This immediately means that this system is restricted to updates within a single version – while the effect of a security patch will be to modify some files, upgrading to a new version would require installing entirely new files. Upgrading from one FreeBSD release to another can already be performed simply by performing a binary install from the published release images.

A more serious issue arises with the kernel: We can only provide updates for the GENERIC kernel. While this may be sufficient for some users, a very obvious class exists for whom this is not sufficient – those with multiprocessor systems. We suggest therefore that it would be a Good Thing if FreeBSD releases also included at least a GENERIC-MP kernel, identical to the GENERIC kernel except for the addition of multi-processor support; indeed, it might be adviseable to add 'bloat' to the GENERIC kernel in the interest of reducing the probability that someone would be required to build a custom kernel – noting, of course, that kernel modules can be updated, so features which can be fully supported via modules would not need to be compiled into the kernel.

Perhaps the most serious issue arises with configuration files and metadata. There are circumstances where a security update might need to change an option in a (user-serviceable) configuration file, or change the ownership, permissions, or flags on a file. This could be handled by transmitting patches for text files rather than simply distributing the new version, and recording which patches have already been applied; changes in ownership, permissions, or flags could be handled in a similar manner. However, such a mechanism would carry with it a considerable risk of damaging a customized configuration; consequently we feel that the principle of least astonishment requires that such (rare) fixes be left up to a human administrator.

A final issue arises from the use of the MD5 message digest. Although no collisions have been found, there is an ongoing attack [NP03]; indeed, it has been recommended for many years that MD5 not be used for applications where collision resistance is required [Ro96]. We note that even given the ability to compute MD5 collisions an attack would be very difficult, since it would require that specially crafted source code be introduced into the security branch; consequently, we consider the risk introduced by using MD5 to be negligible, and justifiable in light of the lack of stronger hash programs in the base FreeBSD distribution.

8 Future work

It seems very likely that the same approach, and much or all of the same code, can be used for building binary security updates to other BSD operating systems. There would, almost certainly, be a different set of patches necessary to remove spurious variabilities; but it would be very surprising if those necessary patches could not easily be found.

On the other hand, for various reasons it seems unlikely that this same approach could be applied to software from the FreeBSD ports tree. First, while pre-built packages are available, most people build ports from source; as noted earlier, this makes it impossible to provide updates. Second, there is no "security branch" for the ports tree; consequently, updating from one version to another is quite likely to involve adding or removing files, which is beyond the scope of this system. Third, the large number of ports, the time necessary for building them, and the (quite common) cases where some ports cannot be successfully built would all contribute to making such an attempt a logistical nightmare. We note, in any case, that portupgrade already makes the updating of ports quite simple.

On the other hand, this tool is ideally suited to self-contained "toasters". Providing that a vendor can construct an automated mechanism for building all installed software (operating system and software), binary updates can be built and distributed in exactly the same manner as for an operating system alone.

One possible future modification would be to keep a 'clean' copy of configuration files, and use mergemaster(8) to merge any changes if the configuration files were changed (in the same manner as when recompiling the entire world). This would not be automated – merging any changes would require the intervention of the system administrator – but it would at least be a step in the right direction.

The most interesting possibility, however, is for having several machines build updates and cross-sign. This would be far from trivial, since each machine would (due to the build stamps) produce different updates (the same files would be modified, but the new values would be different). However, it should be possible for each machine to fetch the updates built by each of the others and remove the build stamps before comparing; in this manner, they could each verify that each others' builds were identical up to (security-irrelevant) build stamps. Client machines could then be configured with a list of trusted keys, and could require that a certain quorum had signed a set of updates before installing. Since, at present, compromising the security of the single machine which is building the updates would allow an attacker to issue trojaned "updates" to a large number of machines, distributing the building process would certainly be advantageous.

Ideally, one would hope that some day the convoluted methods used here will be unnecessary. Software installed from the ports tree, and some other operating systems, have the advantage of being completely packagized; this makes it easy to install or remove specific packages, and consequently makes it trivial to update everything on the system. If FreeBSD were split into such independent packages, a wide range of problems would be simplified; however, performing such a task would most likely require reworking the entire build system, and it seems likely that development will never stop for long enough to make such an effort possible, even if someone could be found with the necessary capability and time to perform such a task.

9 Acknowledgements

The author would like to thank Chad David and Terry Lambert for their assistance in explaining the FreeBSD build process; Nathan Sidwell for his assistance with gcc; and Graham Percival for replacing a dead hard drive, repairing a broken filesystem on a working hard drive, and otherwise helping to maintain the author's FreeBSD box while he was 4700 miles away.

The author would also like to acknowledge support from the Commonwealth Scholarship Commission, which is funding his studies at Oxford University.

10 Availability

The client (update installing) and server (update building) code is available under an open source license from

`http://www.daemonology.net/freebsd-update/`

The client code is also available in the FreeBSD ports tree.

References

[An02] M. Anuzis, *Incident Analysis of a Compromised OpenBSD 3.0 Honeypot*, `http://www.anuzisnetworking.com/whitepapers/obsd30/` (2002).

[Bi02] *FreeBSD Binary Updater Project (binup)*, `http://www.freebsd.org/projects/updater.html` (2002).

[CE03] *CERT Advisory CA-2003-10 Integer overflow in Sun RPC XDR library routines*, `http://www.cert.org/advisories/CA-2003-10.html` (2003).

[Ga03] Gerardo Santana Gómez Garrido, *Binary patches for OpenBSD*, `http://www.openbsd.org.mx/~santana/binpatch.html` (2003).

[Ho02] S. Holcroft, *Incident Analysis of a Compromised RedHat Linux 6.2 Honeypot*, `http://www.holcroft.org/honeypot/Incident/sholcroft-4.1-2002.html` (2002).

[IS03] InstallShield, *InstallShield - Update Service*, `http://www.installshield.com/isus/` (2003).

[La03] D. Langille, Personal email (23 June 2003).

[NP03] *The NEO Project*, `http://www.theneoproject.com/` (2003).

[Pe03] C. Percival, *Naive Differences of Executable Code*, `http://www.daemonology.net/bsdiff/` (2003).

[Po02] J. Polstra, *CVSup*, `http://www.cvsup.org/` (2002).

[Pr03] M. Prettejohn, Personal email (27 May 2003).

[PST03] Puget Sound Technology, *Binary Updates for NetBSD*, `http://pugetsoundtechnology.com/services/netbsd/updates/` (2003).

[Ri92] R. Rivest, The MD5 Message-Digest Algorithm, RFC 1321 (1992).

[Ro96] M.J.B. Robshaw, *On Recent Results for MD2, MD4, and MD5*, RSA Laboratories Bulletin, November 1996.

[Si03] N. Sidwell, Personal email (14 Feb 2003).

[SPW02] S. Staniford, V. Paxson, and N. Weaver, *How to 0wn the Internet in Your Spare Time*, Proceedings of the 11th USENIX Security Symposium (2002).

[SA02] *FreeBSD Security Advisories SA-01:40, SA-01:42, SA-01:48, SA-01:49, SA-01:51, SA-01:52, SA-01:53, SA-01:55, SA-01:56, SA-01:57, SA-01:58, SA-01:59, SA-01:62, SA-01:63, SA-02:08, SA-02:13, and SA-02:25*, `http://www.freebsd.org/security` (2002).