Security of Distribution Mechanisms for Linux and BSD Operating Systems

Gabriel Ewing
Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio

Kevin Nash
Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio

***Abstract*—The abstract goes here.**

I. Introduction

An operating system is a piece of software that manages computer hardware resources and provides a variety of services for computer programs. The central core of an operating system, its kernel, is the first layer above hardware itself. Due to of the depth of their functionality, compromised operating systems can potentially yield a great deal more power to an attacker than application software.

Operating systems can be compromised by malicious computer software, such as rootkits, in the course of normal operation. However, attacks can sometimes be launched more easily against the distribution process itself. This can be done in such a way that users unknowingly install a modified version of the expected operating system. If successful, attacks that result in the distribution of a compromised operating system can be both difficult to detect and powerful.

The open-source software model possesses a somewhat different attack surface that its closed- or shared-source counterparts. Open-source operating systems often have much smaller core development teams than popular commercial systems such as Windows and OS X. Open-source operating systems are rarely distributed using physical media, which is the primary distribution mechanism for Windows. The commercial interfaces that are requisite for online distribution of proprietary software also have advantages and disadvantages in security that are different from the security considerations that open-source distributors must account for.

II. Operating System Distribution

The specific process by which a new version of an operating system travels from developer to end user differs between development teams. Operating system developers establish their own schedule standardizations, though the time between starting the release process and the anticipated release typically takes no more than 90 days. Teams may implement some steps less formally than others do, though the general procedures are ubiquitous enough they can be summarized by a few key steps.

A. System Changes

In an operating system's natural lifecycle, users tend to discover that the system exhibits certain unexpected behaviors. These bugs are reported, tracked, and hopefully fixed in code so that users will not experience them in future versions of the operating system. Users might request that entirely new features be added or that existing, intentional behaviors be modified. Even without user input, changes to the system inevitably originate from within the development team; design paradigms change and the world of security is ever-evolving.

No matter the cause, an operating system's codebase can change frequently. After enough changes are made to warrant a new release, or at certain milestones in a predetermined development schedule, developers stop making new changes in preparation for a code review. The most developers of different operating systems will "freeze" or "lock" the code so that no more changes can be made. Before the code freeze occurs, developers finish integrating their changes and merge the development branch into a stable, or production, branch. Of course, the naming of branches varies between projects. Once the code is frozen, the interested public and, in almost all cases, an internal body begin a code review.

B. Code Review

Following the code freeze, new features are tested for proper functionality, and bug fixes are tested for robustness. In general, the code is reviewed for proper practice and adherence to in-house standards. A beta image is built and distributed to testers. As necessary minor changes are made, such as those to update documentation, fix remaining bugs, and address any discovered security issues, additional beta images will be released for testing.

C. Release Building

Once the beta images are proven to meet quality standards, a branch for the new version is created. The most successful beta images will be included on the branch as "release candidates" a term for beta versions of software that have the potential to be a finished product. The release candidates will undergo further stability testing furher Eventually, a release candidate (not necessarily the most recent) will be selected as the final, stable version.

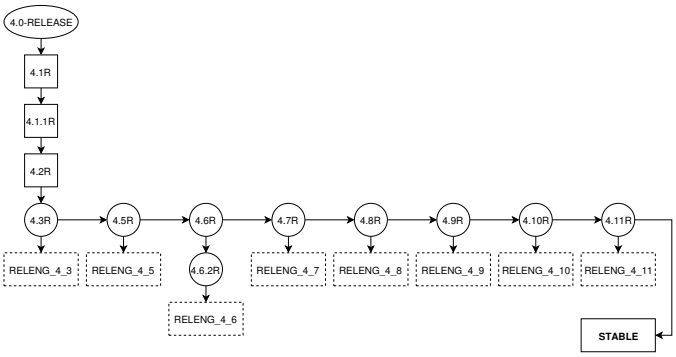

Fig. 1. FreeBSD's RELENG_4 (4.x STABLE Branch)

D. Pre-Distribution

Once the stable release image has been slated for distribution, the mirror sites must be updated to serve this new image. Release engineers will coordinate with the administrators of their mirror sites to ensure that the sites have download access to the new image and that the site is aware of the staged release. Because each mirror site may be managed by an independent administrator, it can take multiple days for all FTP and HTTP sites to reflect the new software.[FreeBSD Release doc section 4.1] The process of coordinating with the mirror sites is a crucial final step on the release team's end, so it is begun several days before the public release day. Once release day arrives, granted a given mirror site has the new operating system, it will announce the public availability of the new software.

E. Image Mirroring

Administrators of mirror sites have their own tasks in ensuring a successful software release. They maintain at least one local copy of the operating system, ensuring that it is kept up-to-date, that its syncronization is error-free, and that their hosting resources are healthy and fully utilized by the public.

1) Mirror Qualifications: The qualifications and requirements for official mirrors of operating systems vary by the system in question, and different areas of distribution differ in their own inherent requirements. For example, an FTP site often represents the largest amount of data that needs to be mirrored; it can contain a live file system, the ISO images of the installation distribution, and branches or snapshots of checked-out source trees. The amount of required space is also multipled by the number of operating system versions and architectures.

All operating system archives demand a certain minimum amount of disk space. It is one of the primary requirements of a mirror. The full distribution can have a size anywhere from 500 gigabytes to multiple terrabytes, depending on the size of operating system, the number of snapshots and releases to be offered, and the number of architectures supported. For example, at time of writing, the Ubuntu archive uses 912GB of disk space and FreeBSD uses 1.4TB for its FTP distribution.

There are a number of system requirements that depend on the expected number of clients. Since official mirrors are designed to handle large volumes of traffic, official mirrors can be required to meet hard standards above the inherent needs of a server. These standards can include minimum CPU speeds, amounts of random access memory, and even certain disk configurations such as RAID.

Finally, operating system authorities might have policies unrelated to hardware, such as requirements about frequency of syncronization. For example, FreeBSD requires that mirrors update once per day, at a minimum. Ubuntu requires that its country-level mirrors update every six hours for archives or every four hours for active releases.

2) Fetching Release: Offering up-to-date versions of an operating system across several mirrors poses the task of ensuring all mirrors host the same content. To accomplish this task, servers commonly use specialized software that synchronizes files between two different systems. rsync is a widely-used utility that functions as both a file synchronization and file transfer program. It minimizes network usage by implementing a type of delta encoding. With this utility, the operators of mirror servers operators can fetch a new ISO image upon release and periodically check for differences (deltas) between an image on the master server and the local copy.

By default rsync compares metadata, specifically the size and modification time, of files. This comparison can be done very quickly, but it is not comprehensive and will not synchronize modifications that do not change this metadata. An alternative behavior can be invoked, however; calling a check with the option `--checksum` will split its file into chunks and compute a set of two checksums for each chunk. The first is a speedy rolling hash and the second is a more traditional MD5 hash. These sums are then passed to the master server, which will send the chunks whose rolling hashes differ from those computed by the calling server. If the rolling hashes match, the master server will compute the slower MD5 hash and compare it against the MD5 hash that it received from the calling server. In this way only parts of a file that have been modified are sent.

The rolling checksum has a length of 32 bits, and MD5 uses a length of 128 bits; combining the two yields an entropy of

$$2^{32+128} = 2^{160}$$

and therefore it is very unlikely that rsync will fail to detect a modification. It is worth mentioning that the purpose of rsync's `--checksum` option is not to ensure cryptographic security but rather to ensure that modifications to master files—imagine those on a frequently changing beta image—are noticed and synchronized without needing to fetch the entire release.

F. Distribution Methods

Linux and BSD distros are usually offered over a variety of protocols, each with separate advantages and disadvantages.

1) HTTP: Body of text goes here...

2) FTP: Body of text goes here...

3) BitTorrent: Body of text goes here...

4) Physical Media: Body of text goes here...

III. Attacks on Distribution

Body of text goes here...

A. "Attack One"

Body of text goes here...

1) Notable Usage: Body of text goes here...

2) Countermeasures: N operating systems currently implement these countermeasures, including Foo, Bar, Baz...

Visual aid goes here

B. "Attack Two"

Body of text goes here...

1) Notable Usage: Body of text goes here...

2) Countermeasures: N operating systems currently implement these countermeasures, including Foo, Bar, Baz...

Visual aid goes here

C. Hash Collisions

Providing a signed checksum is part of the best practices for securing download mechanisms. It allows the user to verify the authenticity of the checksum. Once the user is sure that the checksum is the one that they intended to download, they can use the same checksum algorithm to generate their own hash of the downloaded image. If the downloaded checksum and the generated checksum are identical, the user can then proceed, confident that the image they downloaded is the same one that the distribution maintainers published. However, certain commonly-used checksum algorithms are no longer strong enough to reliably verify the correctness of an image.

Weak hashing algorithms introduce a vulnerability to the distribution process. Usually, an attacker that wishes to replace a published image also has to replace the associated published hash with the hash of the malicious image. Signed hashes alert users to this problem. If the hash of the malicious image is the same as the hash of the original image, though, the attacker does not need to replace the published hash and signing mechanisms are no longer sufficient.

Hash collision attacks are particularly dangerous because normal protection policies do not detect their presence, allowing a malicious image to be installed silently. There are ways to detect a successful hash collision attack: one is that the malicious image would be different upon comparison to the original published image. Another is that distributed source code can be independently built and compared to the malicious image. If the distributed source code has also been replaced, then a code review or version control logs could offer clues. However, none of these checks is part of the standard practice recommended by an distribution that we surveyed. Because of this issue, weak checksum algorithms are a very dangerous practice.

1) Weak Algorithms: The MD5 hash algorithm was published in 1992 by Ronald Rivest [**?**]. Weaknesses in the algorithm were quickly discovered [**?**] and MD5 quickly dropped out of favor among the cryptographic hashing community. In 2004, a collision of the full MD5 algorithm was published [**?**]. MD5 collisions were used in proof-of-concept attacks on certificate authorities as early as 2005 [**?**]. While finding a collision that successfully installs and executes useful malicious tasks is more difficult than simply finding a collision, computing power has increased considerably since 2004 and it is not unreasonable to suspect that this capability exists in the world today.

Unfortunately, many of the of the distributions that we surveyed, including Mint, Ubuntu, Debian, Arch, Slackware, and Mageia, offer MD5 checksums. While none of them offer only MD5, many list it as the first option. MD5 is faster than the SHA variants, and is reliable in detecting random corruption from transmission, but the time to compute a checksum using a strong hash algorithm is still negligible compared to the time to download the image on any modern hardware. We consider offering MD5 checksums a poor practice because users with minimal cryptographic background may choose to only verify using MD5 and trust that a hashing algorithm used by the distribution is secure. Replacing an image and MD5 checksum would remain undetected until a stronger algorithm was tried and the distribution maintainers were notified.

Recently, significant advances have been made towards collisions of the SHA-1 algorithm, and there may be a collision of the full algorithm discovered in the near future [**?**]. This has led to the planned deprecation of SHA-1 Transport Layer Security (TLS) certificates in Google Chrome and other browsers [**?**].

While an impending collision of the full SHA-1 algorithm is concerning, the previously-mentioned problem of finding a colliding image that executes malware is somewhat more difficult. For practical security purposes, using SHA-1 checksums is likely sufficient for the time being.

2) Countermeasures: Distribution maintainers should remove MD5 checksums from their download pages and offer checksums generated by state-of-the-art algorithms. Additionally, they should compare the images offered by mirrors and their main download server to the image that they originally built on a regular basis. Users should verify downloaded images using the strongest algorithms available.

IV. External Risks

Body of text goes here...

A. Re-Hosting and Ownership Hijacking

1) SourceForge: SourceForge is a hosting service for open-source software. In 2015, the website was accused of bundling malware with the binary project packages that it offered for users to download. This caused some large projects to abandon the site entirely [**?**]. While the company later announced a change to this policy [**?**], accusations have continued [**?**] and SourceForge can no longer be considered a trustworthy hosting platform.

Manjaro Linux was the sixth most-popular Linux distribution between March 2015 and March 2016 according to DistroWatch [**?**]. According to our research, SourceForge is the only download platform recommended and offered by Manjaro. While downloads are available via torrent, the torrent links are hosted by SourceForge as well and so the original seed may be of a compromised image [**?**]. We were unable to locate any alternative download mechanism.

This means that there is a real possibility that as of this writing, image downloads of Manjaro are bundled with malware. We recommend that users avoid downloading, installing or running Manjaro Linux until the distribution maintainers review their practices and provide alternatives for downloading.

V. Modern Verification Tools

A. Signify

Recently, Ted Unangst, an OpenBSD developer, created *signify* (*sign* and *verify*), an application for signing messages and verifying signatures. Signify is similar to GPG and PGP, but claims to reduce complexity and improve ease of use [**?**], which has historically been a barrier to widespread PGP adoption [**?**] [**?**]. Signify is built on the Ed25519 elliptic curve algorithm for key exchange in the Diffie-Hellman protocol [**?**].

1) Key Rotation: To use signify to secure image verification, the distribution maintainers hold a private key and the end users hold a public key. The dangers to this architecture are that the private key could be compromised, allowing attackers to sign malicious images, or the user could be given the wrong public key, with the same consequence.

To address the problem of a compromised private key, OpenBSD now generates new public and private keys for each release of the distribution, which are published every six months [**?**]. If a private key is compromised, the attackers will be able to sign malicious images of that release. However, this will be useful for at most six months, as installation of old releases is less common.

Key rotation is also useful for distributing trusted public keys. Since signify was completed, each OpenBSD release image has included the public key for the subsequent release; that is, an OpenBSD 5.6 image has the public key for 5.7, 5.7 has the public key for 5.8, and so on. This means that once the user has a correct image installed, they will be able to recursively and securely download the subsequent release for as long as they wish [**?**].

2) Public Key Trust: The problem remains that the user must still use an untrusted public key when verifying the initial installation. The OpenBSD maintainers have chosen to create a web of trust by displaying the current public key wherever possible. The current keys fit in 56 base64-encoded characters, which is small enough to be converted into a QR code. The public key for OpenBSD 5.7 was shown in that format at the conference where signify was introduced. Other places where they have written the key include OpenBSD release CDs, openbsd.org, and Twitter [**?**].

3) Future: Signify has been ported to Linux [**?**], OS X [**?**], and Windows [**?**]. Because of its ease of use and support by the OpenBSD project, we believe that signify will gain widespread adoption for verifying releases in years to come.

VI. Best Consumer Practices

A. Choosing a Protocol

Body of text goes here…

B. Verifying Mirrors

Body of text goes here…

C. Building a Web of Trust

Body of text goes here…

D. "Soft" Risk Mitigation

Sometimes it is best to rely on proven-stable releases. It can be harmful to be on the bleeding edge of development, although it is a service to the industry.

VII. Our Implementations

Body of text goes here…

VIII. Conclusion

The conclusion goes here.

References