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*—We investigate the issue of the security of distribution of images of open-source operating systems. We review the stages in a common distribution process, possible attacks on the process, and practices of individual distributions. We use work in the literature on cryptographic algorithms to identify algorithms that should not be used in this context. We offer guidelines for how distributions can secure their distribution processes and how consumers can best insulate themselves from the associated risks.**

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.

In the course of research for this paper, we looked at the top ten Linux distributions on DistroWatch [1], plus Slackware Linux, Gentoo Linux, FreeBSD, and OpenBSD. We noted the default installation choices that these distributions offer to users, recent news involving them, and various miscellaneous pieces of information. In this paper, we present and discuss our most interesting findings.

A. Effects of Compromised Operating System

Allowing an attacker to write an arbitrary operating system that is subsequently installed on a server or personal computer offers an extraordinary level of invasion of privacy and system access. Every action that a user takes on a computer goes through the operating system. It would be trivial to write a logger for all network traffic on a computer and periodically send it to the attacker. If the compromised system is a web server, all information stored in the database would be freely available unless encrypted before it reached the server. A laptop with a built-in or external camera could broadcast all visible activities to an attacker. A keylogger could log credit cards and passwords and transmit them periodically. Perhaps the most likely action by an attacker would be to use the malicious operating system as part of a botnet, sending spam emails or carrying out denial- of-service attacks. While these may not directly impact the user, they have a negative effect on the global state of security. Great care should be taken to avoid the installation of compromised operating systems.

B. Conventions

In this paper, we frequently refer to installation images. These are usually .iso files that can be burned to some physical media and installed on a system. Another way to install an open-source operating system is to build the system from source code. Building from source offers the advantage of being able to read code before building an image from it, and can be safer than directly installing an image. Modern operating systems, though, contain millions of lines of code and it is difficult or impossible for any individual or group to conduct a meaningful and comprehensive code review before building from source. This means that many issues affecting binary images also affect source code releases. Unless otherwise specified, when we refer to installation images, we mean that to include source releases.

We investigated operating systems from the Linux and BSD families. We refer to any member of these that is a complete operating system that publishes their own releases as a "distribution", even though this more commonly is used for Linux releases than BSD.

Similarly, we use the phrase "open source" to refer to any piece of software whose source code is publicly available and may be installed at no cost. This includes software whose creators prefer the term "free software".

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. Some teams may implement some steps less formally than others do, though the general procedures are common enough between projects 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 best practices in security as well as trends in operating system design are 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. Then 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 performs a code review.

B. Code Review

Ideally, code is reviewed on a rolling basis throughout the release cycle so that reviewers have time to also deal with any new bugs that come up. In general, the code is reviewed for proper practice and adherence to in-house standards. Code reviewers must be very careful, because detecting subtly inserted malicious code can be a hard task. Many distributions are forced to rely on trust in the code authors to not make deliberate mistakes, and the reviews are only deep enough to catch unintentional bugs.

C. Testing

A beta image is built and distributed to testers. New features are tested for proper functionality, and bug fixes are tested for robustness. 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 [2]. Software development processes and schedules are well-studied and too complex to cover well in this paper.

D. 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. Eventually, a release candidate (not necessarily the most recent) will be selected as the final, stable version [2].

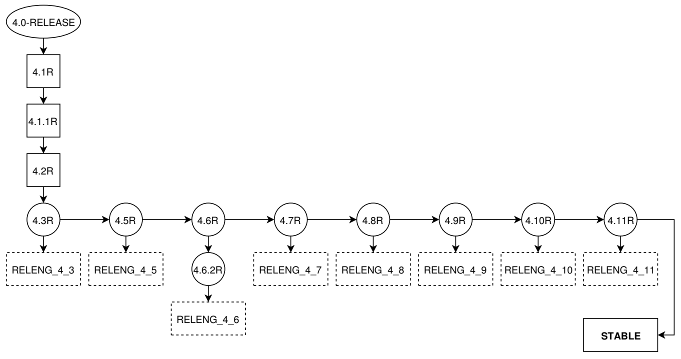


Fig. 1. FreeBSD's RELENG_4 (4.x STABLE Branch)

E. 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. 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. A file enumerating the hashes for each architecture is distributed alongside the installation images themselves. This file is made available via the mirror site, or the hashes are displayed on the web page beside download hyperlinks, or both. Once release day arrives and a given mirror site has the new operating system, it will announce the public availability of the new software [3].

F. 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 synchronization is error-free, and that their hosting resources are healthy and fully utilized by the public [4].

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 images of the installation distribution, and branches or snapshots of checked-out source trees. The amount of required space is also multiplied by the number of operating system versions and architectures [5] [6].

All operating system archives demand a certain minimum amount of disk space. Keeping the archives uncorrupted and on-disk is one of the primary requirements of a mirror. The full distribution can have a size anywhere from 500 gigabytes to multiple terabytes, 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 the time of this writing, the Ubuntu archive uses 912GB of disk space and FreeBSD uses 1.4TB for its FTP distribution [5] [6].

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 [5] [6].

Finally, operating system authorities might have policies unrelated to hardware, such as requirements about frequency of synchronization and refusal of upload connections. The previous two requirements examples of distributions attempting to set some security standards for their mirrors. As an example of a requirement not necessarily directly related to security, 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 [5] [6].

2) Fetching Release: Offering up-to-date versions of an operating system across several mirrors presents the challenge of ensuring all mirrors host the same content. To accomplish this reliably, 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 installation image upon release and periodically check for differences (deltas) between an image on the master server and the local copy [7].

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 [8] [9].

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 [10]. 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 [8].

G. Distribution Methods

Linux and BSD distros are usually offered over a variety of protocols, each with separate advantages and disadvantages. The File Transfer Protocol (FTP) and the HyperText Transfer Protocol (HTTP) are very similar. In the present day, the efficiencies of both are intimately tied to TCP. The analysis of and means by which BitTorrent works are rather different. Transferring physical media is incomparable to any of the other protocols from a network analyst's perspective.

1) FTP and HTTP: FTP is older than HTTP and features a few inefficiencies that HTTP addresses. First, FTP spends three round trips as overhead in getting the first file and two round trips as overhead in getting all subsequent files. An FTP client will open one TCP connection with the server for control messages and another, separate, connection for the file transfer itself [11].

FTP is considered to be slightly slower than HTTP except in certain cases involving the transfer of large files, among which certainly are operating system installation images. FTP transfers do not use encryption, although all of the operating systems covered by our research are offered for free and without any sort of website login, so in these cases, the goal is the raw binary transfer of software and so network sniffing is not a concern.

HTTP overlaps the function of FTP, having been developed following the latter. HTTP will always achieve the best-case overhead of FTP (2 round-trip-time), even upon transfer of the first file. It is the more responsive of the two measuring request-response time of small files and does not require a control channel [11]. HTTP can be used over SSL (HTTPs), which is a substantial advantage over FTP in some cases. However, for the same reasons as above, encryption is not necessary for the transfer of these operating system. We found that none of the operating systems covered by our research had mirrors that served installation image downloads over HTTPs, although the webpages themselves sometimes were served over HTTPs.

2) BitTorrent: The BitTorrent protocol utilizes a peer-to-peer (P2P) network to transfer files to clients. A P2P network allows a large body of hosts to contribute parts of files to new clients. Conceptually, this is very different from the FTP and HTTP protocols, which mediate the transfer of files from one sender to one recipient. There are a number of advantages to the BitTorrent protocol and few disadvantages compared to FTP/HTTP.

Offering software from a BitTorrent source can save a tremendous amount of bandwidth. In the traditional model, a host uses an amount of bandwidth equal to the size of a hosted file—in this case a large ISO—multiplied by the number of times it is transferred. In the BitTorrent model, the bandwidth is distributed over a large number of sources, known as "seeders"; in this case, a client wishing to download the operating system image may do so from a dozen or more sources [12].

A file transfer over BitTorrent might be completed well before the same file can be transferred over HTTP. This is because a torrent is split into many parts and several parts can be downloaded at once from different sources. Since many connections are opened and maintained with TCP during BitTorrent download, the transfer usually begins slow and gradually increases in speed. This is in contrast to HTTP transfers, which occur at a fairly steady rate [12].

BitTorrent is also very resistant to the types of attacks that have been successful on direct download links. As previously mentioned, BitTorrent splits a file into small pieces, 64 KB to 2 MB in size, and stores the hashes and sizes of each piece in the metadata of the `.torrent` file. Whereas a standard method of hashing on an entire file is vulnerable to a collision if an undetectable modification is made, an attack on BitTorrent that relies on hash collisions would need to orchestrate a separate collision for each chunk without a difference in the size of any torrent chunk. Due to the reliable hashes contained in the metadata of the torrent, content poisoning is made very difficult [12].

The most significant disadvantage to BitTorrent comes into play immediately following the release of a new operating system version. When a torrent is first created, it takes some time before the population of seeders can grow. If a malicious `.torrent` file is created alongside the official release, there is a chance that the torrent masquerading as the official torrent will gain popularity. In this way distributing the responsibility of content hosting among a large unknown population is both a strength and a weakness [12].

3) Physical Media: Some distributions, including Slackware Linux, Gentoo Linux, OpenBSD, and FreeBSD, have options to mail the consumer a piece of physical media containing an installation image. While these offers tend to have a price tag, security of items sent through mail is a much different issue from security of packets sent over a network. At least, physical mail can be an excellent redundancy measure in line with the recommendations in VI-B, particularly for corporate entities to whom a cost-covering price is negligible. In the United States, media sent through mail is probably more secure than over a computer network. Certain federal agencies have the power to open and inspect mail, but if such an agency is targeting an individual with replacements of images on mailed compact disks then that individual could soon have larger problems than a compromised operating system.

	HTTP	FTP	Torrent	Physical
Mint	Yes	No	Yes	No
Ubuntu	Yes	No	Yes	Yes
Fedora	Yes	No	Yes	No
Debian	Yes	No	Yes	Yes
Manjaro	Yes	No	Yes	
Mageia	Yes	No	No	No
CentOS	Yes	Yes	Yes	No
Arch	Yes	Yes	Yes	No
Slackware	Yes	Yes	Yes	Yes
Gentoo	Yes	Yes	No	Yes
FreeBSD	Yes	Yes	No	Yes
OpenBSD	Yes	Yes	No	Yes

TABLE I
FIRST-PARTY AVAILABILITY OF DISTRIBUTION METHODS

III. ATTACKS ON DISTRIBUTION

The large number of steps in the distribution process creates a large attack surface. In this section, we describe some of the possible attacks on the process in general terms.

A. *Redirection or Replacement*

All attacks, at some level, involve tricking the user into taking some action that they would not take if they had perfect information. Redirection and replacement attacks do this at the level where the user clicks on a link to download a file. In a replacement attack, the attacker compromises the master file transfer server or mirror, removes the existing image, and puts a malicious one in its place. In a redirection attack, the attacker compromises the web server that the user retrieves the link from and inserts a link to a file transfer server that they control, which serves a malicious image.

1) Notable Usage: In February 2016, Linux Mint, the single most popular Linux distribution according to DistroWatch [1] was the victim of a replacement attack. For part of a day, the official website FTP download link pointed to a server in Bulgaria which was serving a malicious installation image [13]. The attack had no obvious goal, as the modified image did nothing overtly harmful, but Mint is downloaded frequently enough that a determined attacker with a well-curated malicious image could do serious damage to a number of systems if the same situation happened again. In response, the distribution maintainers moved their website to HTTPs, and moved their default verification to SHA-256 and GPG rather than MD5 [14].

2) Countermeasures: The best way to prevent redirection and replacement attacks is to implement a signing protocol such as GNU Privacy Guard (GPG), or signify (discussed further in Section V-A). GPG allows the distribution maintainers to hold a private key, which they sign checksums with. Users then verify that signature with a public key, and ensure that it matches the checksum calculated on the image that they downloaded. If GPG is implemented and used correctly, it can significantly reduce the impact even if an attacker successfully performs a redirection or replacement attack.

Distribution maintainers and mirror hosts can also reduce the chances of a successful redirection or replacement attack by following good operational security practices, such as segmenting the network that the master and web servers reside on, using strong passwords for the servers, etc.

B. 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 [15]. Weaknesses in the algorithm were quickly discovered [16] and MD5 quickly dropped out of favor among the cryptographic hashing community. In 2004, a collision of the full MD5 algorithm was published [17]. MD5 collisions were used in proof-of-concept attacks on certificate authorities as early as 2005 [18]. 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 [19]. This has led to the planned deprecation of SHA-1 Transport Layer Security (TLS) certificates in Google Chrome and other browsers [20].

An impending collision of the full SHA-1 algorithm is concerning for various security purposes. However, using such a collision in an attack on an operating system distribution mechanism takes more work. Because of the previously-mentioned problem of finding a colliding image that successfully builds and executes malware, an attacker with a tool to generate collisions for a given input would have to go through a very large number of collisions before they encountered one that they could actually use. 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

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 [21]. While the company later announced a change to this policy [22], accusations have continued [23] 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 [1]. 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 [24]. 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 [25], which has historically been a barrier to widespread PGP adoption [26] [27]. Signify is built on the Ed25519 elliptic curve algorithm for key exchange in the Diffie-Hellman protocol [25].

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 [28]. 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 [25].

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 [25].

3) Future: Signify has been ported to Linux [29], OS X [30], and Windows [31]. 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

Security of distribution can be approached from two directions. Distribution maintainers are the focal points of the process: they write the software, build the images, choose transfer protocols, and decide what signing options are available, along with numerous other considerations. The end user, though, still has agency in ensuring that the process is followed correctly and in maximizing their safety within the constructs that the distribution maintainers provide. In this section, we provide recommendations for how consumers can use the available options to protect themselves from attacks.

A. Using Existing Mechanisms

First and foremost, security can be achieved by following the download and verification steps recommended by the distribution maintainers, if offered. While the verification protocols provided are not always perfect, they are a reflection of the areas that the maintainers have chosen to focus on and support. An attack on a recommended protocol is also more likely to be discovered and reported by a third party, because other users will also tend to follow the recommendations if they choose to take verification steps.

Signing protocols such as GPG and signify should always be used if they are available. We do not recommend using a distribution that has taken no steps to provide signatures for their releases.

While researching the current state of this field, we came up with certain ad-hoc, *additional* ways that users can reduce the chance of installing a compromised image. These methods, which we present in the remainder of this section, are not intended as a replacement for standard protocols and do not guarantee security even in conjunction with standard protocols. However, we believe that they will not interfere with the standard protocols and have a real chance of uncovering attacks that standard protocols fail to prevent.

B. Redundancy

1) Mirrors: When downloading an image, a user only needs to use one mirror because that mirror is capable of delivering everything that they need to install the operating system. However, the existence of alternative mirrors offers a usually-untapped potential for improving security, based on the principle that it is more difficult to compromise several servers than to compromise one server.

When choosing to draw from multiple mirrors, there are certain practical tradeoffs to consider. From a security perspective, the optimal choice is to download an image and signed checksums from each mirror. Downloading the image could allow a user to detect a hash collision attack on an individual mirror. Comparing the signed checksums could alert the user to a recent public key replacement if the mirrors update at different intervals.

The improved security has to be balanced against the time spent doing the additional verification. Downloading a full image from each mirror would approximately multiply the total download time by the number of mirrors one is using. Since the network transfer time is often a significant bottleneck in the process of an installation, this may be a real impediment to usability. Weighing the costs and benefits is the responsibility of the individual user.

2) Transfer Protocol: Similarly, redundancy in the transfer protocol used to download the installation image can be helpful. Many mirrors offer downloads over at least FTP and HTTP. While these protocols have similar levels of security, it increases the workload of an attacker to compromise both protocols and so may provide a practical security benefit. Downloading via BitTorrent has a significantly different attack surface from FTP and HTTP, so that should definitely be used if it is an official option from the distribution. Ideally, users should try to combine at least one of HTTP and FTP with a BitTorrent download.

Users can also bias their distribution choices towards distributions that serve download links over HTTPs rather than HTTP, reducing the risk of falling victim to a man-in-the-middle attack. A BitTorrent link served over HTTPs is a very secure way to download an image, provided that the user verifies a sufficient number of torrent seed hosts.

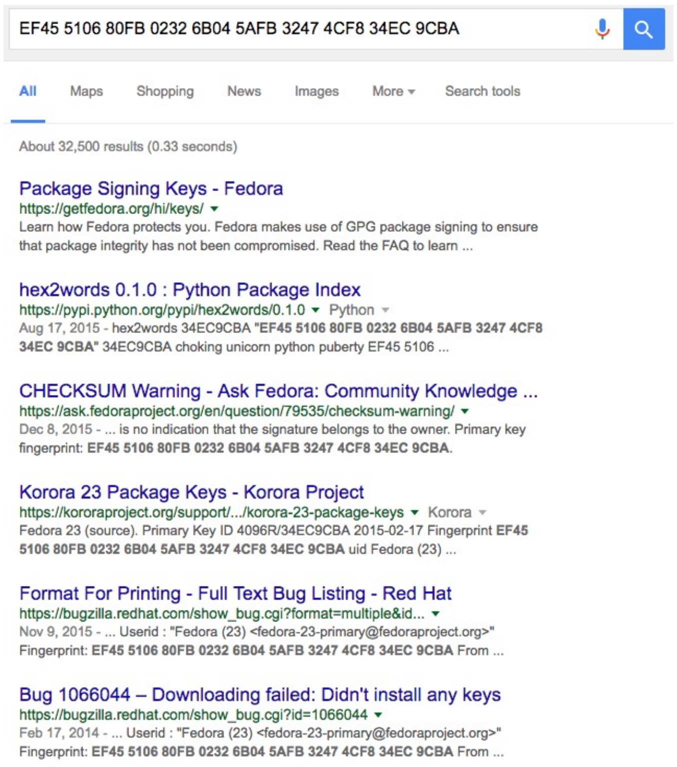
3) Hashing Algorithms: Another potential dimension of redundancy is to compare all provided checksum algorithms. An attacker finding a hash collision for SHA-256 is not completely impossible, but an image that has both a SHA-1 and a SHA-256 collision is even less likely. Again, such methods do not provide a guarantee of security, but have the potential to frustrate the efforts of a determined attacker.

C. Building a Web of Trust

As discussed in Section V-A2, signing protocols such as signify and GPG only ensure that a message was signed by the provider of a public key. For a first-time user of a distribution, the public key is usually taken from the distribution website. A conscientious user should take steps to verify the authenticity of public key beyond the fact that it was retrieved from an official source.

Public keys are often copied to places other than the official website. Intuitively, it is generally more difficult for an attacker to replace a file on the official server plus several third party servers than it is to replace it only on the official server. Therefore, one reasonable way to build trust in a key is to look it up in a search engine such as Google. For popular distributions, such a search commonly results in links to the official website, tutorials, non-security bug reports and other benign items. Older links also help to build trust because they imply that the provided key is the same one that has been available on the official website for a significant period of time. By contrast, finding no links or only links from newly-established websites indicates that there is a problem. For less popular distributions, minimal results should be expected by the search is worth performing regardless. Any recent news articles returned by the search should be thoroughly investigated, as they may provide information about an ongoing compromise.

Figure 2 shows an example of a positive search. The third result, titled “CHECKSUM Warning”, initially may look suspicious but is a question regarding the output of a standard part of the key verification process.


EF45 5106 80FB 0232 6B04 5AFB 3247 4CF8 34EC 9CBA

About 32,500 results (0.33 seconds)

Package Signing Keys - Fedora
https://getfedora.org/hi/keys/
Learn how Fedora protects you. Fedora makes use of GPG package signing to ensure that package integrity has not been compromised. Read the FAQ to learn ...

hex2words 0.1.0 : Python Package Index
https://pypi.python.org/pypi/hex2words/0.1.0 Python
Aug 17, 2015 - hex2words 34EC9CBA "**EF45 5106 80FB 0232 6B04 5AFB 3247 4CF8 34EC 9CBA**" 34EC9CBA choking unicorn python puberty EF45 5106 ...

CHECKSUM Warning - Ask Fedora: Community Knowledge ...
https://ask.fedoraproject.org/en/question/79535/checksum-warning/
Dec 8, 2015 - ... is no indication that the signature belongs to the owner. Primary key fingerprint: **EF45 5106 80FB 0232 6B04 5AFB 3247 4CF8 34EC 9CBA**.

Korora 23 Package Keys - Korora Project
https://kororaproject.org/support/.../korora-23-package-keys Korora
Fedora 23 (source). Primary Key ID 4096R/34EC9CBA 2015-02-17 Fingerprint **EF45 5106 80FB 0232 6B04 5AFB 3247 4CF8 34EC 9CBA** uid Fedora (23) ...

Format For Printing - Full Text Bug Listing - Red Hat
https://bugzilla.redhat.com/show_bug.cgi?format=multiple&id...
Nov 9, 2015 - ... Userid : "Fedora (23) <fedora-23-primary@fedoraproject.org>"
Fingerprint: **EF45 5106 80FB 0232 6B04 5AFB 3247 4CF8 34EC 9CBA** From ...

Bug 1066044 – Downloading failed: Didn't install any keys
https://bugzilla.redhat.com/show_bug.cgi?id=1066044
Feb 17, 2014 - ... Userid : "Fedora (23) <fedora-23-primary@fedoraproject.org>"
Fingerprint: **EF45 5106 80FB 0232 6B04 5AFB 3247 4CF8 34EC 9CBA** From ...

Fig. 2. Google search results for Fedora 23’s public key fingerprint

D. Patience and “Soft” Risk Mitigation

Many users install a new operating system immediately after it is released. We recommend that a security-conscious individual download the image, signed checksum and keys immediately and then wait for some period of time. If the image, signature or keys were replaced by an attack before release, then the time when it is most likely to be noticed is when people are looking around the new system for new features, bugs and other anomalies. After waiting, the user can download everything again and ensure that all the items match.

VII. Automation

Initially, we planned to write scripts to automate our recommendations of best practices for several distributions. However, over the course of our research, we decided that this would likely have a negative impact on security. One reason for this is the fragility of such a script. For example, to automate downloading files from several mirrors for Fedora Linux, we would either have to hard-code the mirror links or scrape the HTML from the official mirrors list. Hard-coded links would be unreliable and eventually break. Scraping the HTML would break when the format of the mirrors list changed. Either of these options would be inferior to a person applying common sense and reasoning to the same process. Once we had the correct server link, we would also have to append a hard-coded file path that contains the release version number. The version number would change at the next version release and the file path could as well. Another problem is that analyzing Google search results is fairly easy for a person but a script would be unreliable at best. We posit that downloading an image for a new operating system is an infrequent and important enough task that the person wishing to do so should devote their full attention to the task.

VIII. Conclusion

Image and source distribution creates an attack surface for any operating system. Some maintainers of open source distributions appear to be neglecting this issue and, in the process, exposing their users to potential harm. Other groups are paying attention and following best practices, and a few are even trying to push the field forward. We found ways that consumers can proactively protect themselves and identify distributions with signs of bad practices. Risk can be seriously mitigated on an individual level with some caution and patience.

References

[1] "DistroWatch page hit ranking," 2016, accessed April 2016. [Online]. Available: https://distrowatch.com/dwres.php?resource=popularity

[2] "Release process," accessed April 2016. [Online]. Available: https://www.freebsd.org/doc/en/articles/releng/release-proc.html

[3] "Distribution," accessed April 2016. [Online]. Available: https://www.freebsd.org/doc/en/articles/releng/distribution.html

[4] J. Kuriyama, V. Vaschetto, D. Lang, and K. Smith, "Mirroring FreeBSD," May 2015, accessed April 2016. [Online]. Available: https://www.freebsd.org/doc/en_US.ISO8859-1/articles/hubs/

[5] P. Collins, "Mirrors," April 2016, accessed April 2016. [Online]. Available: https://wiki.ubuntu.com/Mirrors

[6] "Requirements for FreeBSD mirrors," accessed April 2016. [Online]. Available: https://www.freebsd.org/doc/en_US.ISO8859-1/articles/hubs/mirror-requirements.html

[7] S. Leinen, "Rsyncmirror," November 2013. [Online]. Available: https://help.ubuntu.com/community/Rsyncmirror

[8] J. Morais, "Zsynccdimage," February 2015, accessed April 2016. [Online]. Available: https://help.ubuntu.com/community/ZsyncCdImage

[9] "rsync," accessed April 2016. [Online]. Available: https://download.samba.org/pub/rsync/rsync.html

[10] A. Tridgell, "Rolling checksum," November 1998, accessed April 2016. [Online]. Available: https://rsync.samba.org/tech_report/node3.html

[11] J. Touch, J. Heidemann, and K. Obraczka, "Analysis of HTTP performance," August 1996, accessed April 2016. [Online]. Available: http://www.isi.edu/lsam/publications/http-perf/#intro

[12] X. Lou, K. Hwang *et al.*, "Adaptive content poisoning to prevent illegal file distribution in p2p networks," *IEEE Transaction on Computers*, 2006.

[13] C. Lefebvre, "Beware of hacked ISOs if you downloaded Linux Mint on February 20th!" February 2016, accessed April 2016. [Online]. Available: http://blog.linuxmint.com/?p=2994

[14] ——, "Monthly news-february 2016," March 2016, accessed April 2016. [Online]. Available: http://blog.linuxmint.com/?p=3007

[15] R. Rivest, "The MD5 message-digest algorithm," 1992.

[16] B. den Boer and A. Bosselaers, "Collisions for the compression function of MD5," in *Advances in CryptologyEUROCRYPT93*. Springer, 1993, pp. 293–304.

[17] X. Wang, D. Feng, X. Lai, and H. Yu, "Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD." *IACR Cryptology ePrint Archive*, vol. 2004, p. 199, 2004.

[18] A. K. Lenstra, X. Wang, and B. de Weger, "Colliding X. 509 certificates," Tech. Rep., 2005.

[19] M. Stevens, P. Karpman, and T. Peyrin, "Freestart collision for full SHA-1," *Cryptology ePrint Archive*, no. 2015/967, pp. 1–21, 2015.

[20] L. Garron and D. Benjamin, "An update on SHA-1 certificates in Chrome," December 2015, accessed April 2016. [Online]. Available: https://security.googleblog.com/2015/12/an-update-on-sha-1-certificates-in.html

[21] M. Schumacher, "GIMP project's official statement on SourceForge's actions," May 2015, accessed April 2016. [Online]. Available: https://mail.gnome.org/archives/gimp-developer-list/2015-May/msg00144.html

[22] "Third party offers will be presented with opt-in projects only," accessed April 2016. [Online]. Available: https://sourceforge.net/blog/third-party-offers-will-be-presented-with-opt-in-projects-only

[23] "SourceForge hijacks the Nmap SourceForge account," 2015, accessed April 2016. [Online]. Available: http://seclists.org/nmap-dev/2015/q2/194

[24] "Download Manjaro," accessed April 2016. [Online]. Available: https://wiki.manjaro.org/index.php?title=Download_Manjaro

[25] T. Unangst, "signify: Securing OpenBSD from us to you," in *Presentations: BSDCan 2015*. OpenBSD, 2015, http://www.openbsd.org/papers/bsdcan-signify.html.

[26] A. Whitten and J. D. Tygar, "Why Johnny can't encrypt: A usability evaluation of PGP 5.0." in *Usenix Security*, vol. 1999, 1999.

[27] S. Sheng, L. Broderick, C. A. Koranda, and J. J. Hyland, "Why Johnny still can't encrypt: evaluating the usability of email encryption software," in *Symposium On Usable Privacy and Security*, 2006, pp. 3–4.

[28] "Frequently asked questions," accessed April 2016. [Online]. Available: http://www.openbsd.org/faq/faq1.html#Next

[29] C. Neukirchen, "outils," https://github.com/chneukirchen/outils, 2016.

[30] J.-P. Ouellet, "signify-osx," https://github.com/jpouellet/signify-osx, 2015.

[31] T. Stoeckmann, "signify-windows," https://github.com/stoeckmann/signify-windows, 2016.