Resilient Software Mirroring With Untrusted Third Parties *

Håvard Johansen Dag Johansen
University of Tromsø, Norway
haavardj,dag@cs.uit.no

Abstract

Open-source communities depend on donated third-party servers, known as mirrors, to distribute their software to millions of end-users. However, existing mirror infrastructures lack the mechanisms to deal with the wide-range of faults that can occur. In this paper we describe our ongoing work to construct a mirror infrastructure that is highly resilient to failures. In particular, our infrastructure is constructed to tolerate Byzantine failures so that a potential attacker cannot deny user service even if he is in control of one or more mirrors.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed applications; D.4.5 [*Reliability*]: Fault-tolerance; K.6.5 [*Security and Protection*]: Unauthorized access

General Terms Reliability, security, design

Keywords Intrusion-tolerance, software distribution, overlay network, denial-of-service attack

1. Introduction

Open-source communities depend on third-party servers to distribute their software. This distribution mechanism is commonly referred to as software mirroring. Many large open-source projects, such as the Linux Kernel Archive, the Debian Linux project, and the Ubuntu Linux project are distributed this way. Millions of end-users, organizations, and enterprises rely on these services for acquiring the software components they need, or for acquiring important software updates that fix critical software bugs and vulnerabilities.

Mirror sites are commonly operated by volunteering organizations and individuals that donate spare hosting capacity on their web and file servers to the task of distributing the software. For instance, the Linux Kernel Archive is currently being distributed through 87 mirrors located in 37 different countries or territories. Operators include educational institutions, like the University of Wisconsin, and commercial actors, like the Chicago based company SingleHop[1]. Other large open-source communities, like the Debian and Ubuntu Linux distributions, are similarly dependant upon such donated third-party hosting capacity.

Unfortunately, existing mirror infrastructures are surprisingly fragile as they implement few mechanisms for dealing with faults. For instance, the Debian mirror mailing list[2], which is the Debian organization's main venue for maintaining its pool of mirrors, is riddled with observed errors that must be mitigated manually by mirror operators. For some users, such errors are only a nuisance. However, these errors may also pose a hazard in the case where end-users are dependant on the ability of downloading new software components on demand, for instance to open received e-mail attachments, or to receive critical software updates.

More alarmingly, by pretending to be a benevolent donor running a mirror for the benefit of the open-source community, an attacker might get accepted as an official mirror by some software vendor. In this case, the attacker has intruded into that software vendor's distribution channel enabling him to induce malicious arbitrary (Byzantine) faults within the system. Although digital signatures are used by most software vendors to prevent illicit modifications to the distributed software, having gained control of one or more mirrors, an attacker can potentially slow down or stop the service. Such attacks are particularly hazardous when software security updates are distributed since an attacker can potentially reverse-engineer published security patches in order to construct malware that targets the vulnerable code, and then exploit any client that connects [2, 3, 8].

In this paper, we outline our ongoing effort to construct a software mirror infrastructure that is highly resilient to failures. In particular, we want our infrastructure to be Byzantine fault-tolerant so it is able to deal with the many types of failures that can occur, and so that it can survive poten-

* This work is supported in part by the Research Council of Norway through the National Center for Research-based Innovation program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HotSWUp'08, October 20, 2008, Nashville, Tennessee, USA.
Copyright © 2008 ACM ISBN 978-1-60558-304-4/08/10. . . $5.00

[1] Source: `http://www.kernel.org/mirrors/`.

[2] `http://lists.debian.org/debian-mirrors/`

tial attacks by malicious mirrors. The key idea is to organize the mirrors in an overlay-network structure where each mirror selects multiple mirrors at random as partners. The number of partners is adjusted to strike a balance between fault-tolerance and scalability. Also, by layering our two-phase dissemination protocol [12] on top of such a structure, we can reduce the time in which a security-update derived exploit remains effective, to the time it takes to distribute a small cryptographic key.

2. System architecture

Software mirror infrastructures distinguish three roles: the software vendor, mirrors, and clients. The *software vendor* publishes his software, and subsequent updates, on a *master server* for clients to download and install. The set of all software published by a vendor is referred to as his *software repository*. The vendor may update his software repository by adding, updating, and removing software packages.

For scalability reasons, we cannot rely on all clients downloading all software packages directly from the master server. Instead, the vendor does not distribute his software directly to the clients, but to the collection of *mirrors*.

Each software mirror maintains a read-only replica of the vendor's software repository. Mirrors stay up-to-date by receiving updates from the master server or from some other mirror. The software vendor typically lists mirror sites on his web-page, kindly asking clients to use those in addition to the master site when downloading software. The mirrors are then responsible for providing the clients with software packages upon request.

We assume that the software vendor is benign and that he is trusted by both the clients and the mirrors. In particular, using public-key cryptography, mirrors and clients can ascertain the authenticity of the software they download. Mirrors are not trusted as they are administrated by third-parties. As such, each mirror has a state that is either *correct*, *crashed*, or *Byzantine*. Correct mirrors faithfully execute the specified protocol, while crashed mirrors do not execute any protocol steps. Byzantine mirrors are not bound by the protocol and may execute arbitrary instructions. We refer to mirrors that are either correct or Byzantine as *live*, and mirrors that are either crashed or Byzantine as *faulty*.

Correct mirrors might be unreachable and appear crashed to other mirrors due to transient network outages. Byzantine mirrors can disguise themselves as correct mirrors by executing the protocol, or as crashed mirrors by not executing at all. Hence, we cannot, in general, determine which mirrors are Byzantine unless they reveal themselves as such by sending messages that prove that they are misbehaving.

An attacker might gain control of a mirror by, for instance, donating a server or by hacking a correct mirror. However, we do assume that the fraction of Byzantine mirrors amongst all live mirrors can be probabilisticilly upper bounded by a certain P_{byz}. The need for such an upper bound is an unfortunate but well established necessity [6, 14]. Clients are passive participants, and in particular do not participate in the distribution of software. Thus, we do not have to make assumptions on the correctness of clients.

3. Resilient software distribution

A software repository is a collection of software package files organized in some directory structure. For instance, in Debian Linux a software package is a *deb* archive file, which can be downloaded and installed individually by the clients. Debian releases both source software packages, which can automatically be compiled by the clients, and pre-compiled binary packages for a wide-range of computer architectures. The Debian software repository contains a *manifest file* listing all available software packages and their secure hash and size. The clients typically use only a small subset of all available software. The manifest file, and other meta-data, is used by the clients to select the packages they want to install and download.

In existing mirror infrastructures, clients are typically configured to receive software from a single mirror. If a client c selects a faulty mirror as a software source, then c is subject to the following two key problems:

- Whenever the software vendor publishes a new software package p, he updates the manifest file to include information about p. If the faulty mirror does not forward the updated manifest file to c, then c will not learn about p. Consequently, c will not be able to download and install p. Although, having clients download manifest files directly from the master server solves this problem, this is not an ideal solution as we want to minimize the server infrastructure that the software vendor needs to maintain.
- If c has received the updated manifest file and tries to download p, the faulty mirror might delay or stop c's reception of p by throttling or halting the download or by sending corrupt data.

Clients can mask these faults by using multiple mirrors as sources for software. For instance, in the case where c knows about p, c can simply retry the download using different mirrors until p is successfully received. However, for distributing manifest files, clients need a mechanism to determine when a sufficient number of mirrors have been used in order to distinguish the case where a mirror is faulty from the case where no new manifest file is available. We will describe such a mechanism next.

3.1 Locating a correct mirror

We instrument each client such that they pick at least k mirrors at random from the set of all live mirrors, and use those to receive information about available software packages. By adjusting the value for k, we can increase the likelyhood that at least one of the selected mirrors is correct. However, we do not want to pick k too large, as this would generate un-

necessary load on the mirrors. Instead, we want to choose the minimal value for k so the probability of a client having only faulty mirrors becomes smaller than some configured target fault-tolerance level ϵ.

Due to the randomization of mirror selection and the probabilistic upper bound on the fraction on Byzantine mirrors among all live mirrors, P_{byz}, the probability of a client having selected only faulty mirrors is ${P_{\text{byz}}}^k$. Hence, we can find k by calculating:

$$\min_k : \epsilon > {P_{\text{byz}}}^k$$

For instance, given a fraction $P_{\text{byz}} = 0.10$ of faulty mirrors and a target fault-tolerance level of $\epsilon = 10^{-7}$, then each client should connect to at least $k = 7$ random mirrors.

3.2 Mirror overlay network

For scalability reasons, it is not sufficient to offload the clients from the master server to the pool of mirrors. As the number of mirrors grows, we must also prevent the master site from being overloaded with requests from the set of mirrors. The Debian organization has already realized this and divides its mirror into two classes: primary and secondary mirrors. Primary mirrors receive updates directly from the master server. Secondary mirrors receive updates from either a primary mirror or some other secondary mirror. Still, mirrors select only one upstream source to receive updates from. Although, if kept balanced, diffusion trees distribute load efficiently, they do not tolerate failures well. For instance, if a primary mirror m fails, all secondary mirrors receiving updates from m will be affected. In our case, we are also concerned with a potential intruder attempting to delay or stop the distribution of critical software updates.

To tolerate Byzantine faults, we organize the mirrors such that each mirror download updates from multiple mirrors in a peer-to-peer like manner. For this, each mirror m chooses t random partners from the set of all live mirrors and connects to those. However, unlike with clients, it is not sufficient to choose t such that m will have at least one correct mirror with high probability. The resulting mesh of mirrors must contain sufficient redundancy and diversity such that, with high probability, the subgraph of correct mirrors are connected, and that this subgraph includes the master server. As there is an overhead associated with each partner that m receives updates from, we want t to be as small as possible.

We can obtain a value for t using the classic result of Erdös and Rényi [7], stating that in a random graph of n nodes, if the probability of two nodes being connected is $p_n = (\log n + c + o(1))/n$, then the probability of the graph being connected goes to $\exp(-\exp(-c))$. In our case, the number of correct mirrors, n, is expected to be at least $(1 - P_{\text{byz}}) \times N$, where P_{byz} is the upper bound on the probability that a live mirror is Byzantine and N is the total of the correct and the Byzantine mirrors. Then the probability that one mirror is connected to another is $1 - (1 - 1/N)^t \approx t/N$. Thus $p_n \approx 2t/N$. In order for the correct mirrors to be connected with probability γ, we obtain

$$t \geq \frac{N}{2n} \cdot \left(\log \frac{-n}{\log \gamma} + o(1) \right)$$

For instance, given a fraction $P_{\text{byz}} = 0.10$ of a mirror being faulty and a target probability of $\gamma = 1 - 10^{-7}$, then each mirror should connect to at least 13 randomly selected mirrors.

3.3 Access control

If an attacker is able to gain control of an unbounded number of mirrors, the ability of a distributed system to mask Byzantine faults by means of redundancy is undermined [6]. To make it difficult for an attacker to enlist a large number of mirrors under his control, we are deploying a simple mirror access control mechanism based on public-key certificates. Currently, we use commonly available public-key management tools that are compliant with the X.509 standard.

To be accepted as a mirror, a server administrator must first obtain a mirror certificate from the software vendor. Upon receiving a request for such a certificate, the software vendor must check the validity of the requesting site. For instance, he could check that the requesting entity is indeed a valid institution or individual, and that it seems reputable. Access control is enforced by instrumenting correct clients and mirrors to not accept connections from mirrors without a valid mirror certificate. The list of valid mirror certificates is distributed inside the software repository and through the software vendor's web-page.

Note that this scheme does not prevent an attacker from entering the pool of mirrors as the software vendor may make mistakes or be tricked into signing certificates for malicious sites. However, it allows the vendor to limit the rate at which the attacker can do so. As argued in Section 2, we must assume that the fraction of Byzantine mirrors among live mirrors can be kept below the configured probabilistic upper bound P_{byz}.

3.4 Maintaining up-to-date membership information

In order for our mirror selection schemes to work, mirrors and clients must have an up-to-date list of all live mirrors. We are not aware of any open-source communities that have yet automated this task. For instance, the Debian community maintains a web-page that lists all available mirrors. This page is manually updated based on information that mirror operators post on a dedicated mailing list. Clearly, a manual membership maintenance mechanism does not scale well, is slow, and is prone to errors. Also, if an attacker can wrongly modify the membership views of correct mirrors by targeting the underlying membership protocol, he may gain complete control of our infrastructure [6].

To maintain the list of mirrors, we instrument mirrors to use our Byzantine fault-tolerant membership protocol, *Fireflies* [11]. This protocol ensures, with high probability, that

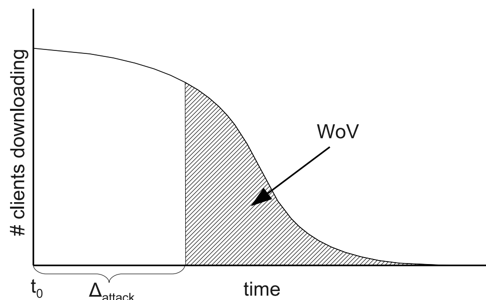

Figure 1. Cleartext dissemination

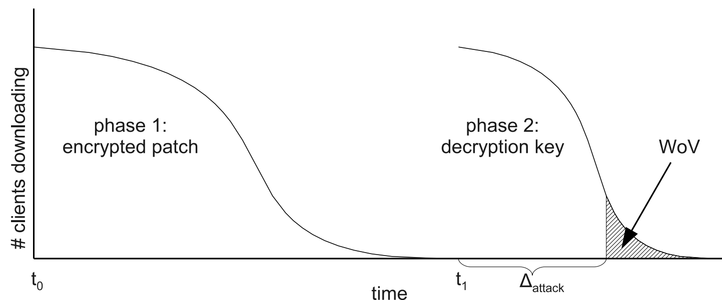

Figure 2. Two-phase dissemination

faulty mirrors cannot keep crashed mirrors in the view of live mirrors, or live mirrors out of these views. For this, mirrors monitor one another using an adaptive pinging protocol, and issue *accusations* (failure notices) whenever a mirror is suspected to have failed. If a mirror is falsely accused, it has the opportunity to issue a *rebuttal* before it is removed from the views of correct mirrors, thus preventing faulty mirrors from incorrectly modifying the membership views.

3.5 Two-phase dissemination

Because software security updates contain information about vulnerabilities, they can be reverse engineered into exploits. Tools for doing this already exist [3, 8]. Consequently, there is a race between hackers and clients to obtain software updates first.

Since we cannot in general distinguish a malicious mirror from a benign mirror, we must assume that the attacker will have access to each update u as soon as the first upload of u from the master server has completed. If t_0 is this time and $\Delta_{\texttt{attack}}$ is the time needed by an attacker to reverse engineer u into an exploit, then, as illustrated in Figure 1, there is a *window of vulnerability* (WoV) that opens at time $t_0 + \Delta_{\texttt{attack}}$ and closes when the number of clients that have not yet downloaded and installed the update, shrinks to insignificance.

In order to reduce the opportunity for an attacker to exploit clients by reverse-engineering software updates, we are instrumenting our mirror infrastructure to make use of our two-phase dissemination scheme. This makes the size of the WoV fixed and small despite the fact that voluminous data has to be transferred over the wire [12].

The idea is to disseminate security updates in two phases. In phase one, we distribute an encrypted version of the software update. The decryption key is known only to the software vendor, such that any attacker receiving u cannot start reverse engineering it. Thus, the WoV remains closed. In phase two we disseminate the decryption key. The advantage of this scheme is, as illustrated in Figure 2, that the WoV does not open until $t_1 + \Delta_{\texttt{attack}}$, where t_1 is the time when phase two starts. As such, the size of the WoV only depends on the time it takes to disseminate a small fixed-size decryption key.

4. Implementation issues

Demanding that mirror operators make vast changes to their existing server infrastructure will most likely reduce the good-will of mirror operators, which open-source communities are so dependant upon. We also expect that some mirror operators will be slower than others to deploy new functionality. It is therefore important the any changes to existing mirror infrastructures are done incrementally and without breaking backward compatibility. Fortunately, the tools used for file mirroring in most large open-source projects lend themselves well to incremental modifications as they are segmented into the following three functional components, which we can modify individually:

- A *local filesystem* where the mirrored software repository is stored.
- A *file-server* configured to serve software repository files from the local file system to both mirrors and clients. Typically, these are HTTP or FTP servers.
- A *script* that runs periodically to check for and download updates.

This design is also attractive from a security point-of-view because it allows administrators to host a software mirror using existing server infrastructure and security policies. It also allows the mirror operator to limit the code dealing with inbound network requests to well tested software, like the Apache web server, which already benefits from an active development community. We are therefore constructing our mirror infrastructure tools so that it follows this design.

The client-side software for selecting, downloading, and installing software packages is typically part of the software repository. As such, updating this software is unproblematic as we can implement the needed changes and distribute the updated software through the mirror infrastructure.

5. Related work

With approximately 300 million clients, Microsoft Windows Update is currently the world's largest software update service [10]. The service consists of a (presumably large) pool of servers that clients periodically pull for updates. Unlike mirrors in open-source communities, Microsoft update

servers are trusted sites. Even so, an attacker might try to gain control of one or more servers by, for instance, installing Trojan programs or exploiting software vulnerabilities [1]. It is unclear how Microsoft protects its servers from such intrusions.

Several peer-to-peer content distribution systems, like SplitStream and Bullet [4, 13], have been constructed to increase the upstream bandwidth of a publisher by spreading data forwarding load to all recipients. However, these systems cannot be used in critical infrastructures, like software mirroring services, as they do not tolerate Byzantine faults.

Fu et al. proposes to use a read-only variant of the Secure File System (SFS) for software mirroring [9]. However, SFS assumes a static membership and does not tolerate omission attacks. Also, the SFS requires specific kernel-level functionality that might not be available on all mirror servers.

Existing Byzantine fault-tolerant state-machine replication protocols, like the BFT protocol [5], can potentially be used for constructing resilient mirroring services. Unfortunately, the overhead of consensus makes these protocols unable to scale to the hundreds of mirrors in use by many open-source communities. Fortunately, the problem of diffusing software updates is weaker than that of traditional state machine replication as mirrors do not have to establish agreement on the sequence of operations to execute. Also, all software updates are signed with the software vendor's private key, so we can avoid the overhead of establishing their authenticity using redundancy [15].

6. Concluding remarks

Open-source communities are dependent on third-party mirrors to distribute their software. However, existing mirror infrastructures are surprisingly fragile as they implement few mechanisms to deal with faults. In particular, we are concerned that an attacker can intrude into the pool of mirrors simply by hosting a mirror site and subsequently induce arbitrary failures. By doing so, an attacker can deny users service and delay the distribution of critical security updates.

In this paper we have outlined our ongoing effort for constructing a resilient software mirror infrastructure. To deal with the wide variety of faults that might occur, and to deal with intruders in particular, we are constructing our mirroring service such that it can tolerate Byzantine faults. These ideas are based on our earlier experience with intrusion-tolerant overlay networks and patch distribution [11, 12].

References

[1] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of Vulnerability: A case study analysis. *IEEE Computer*, 33(12):52–59, December 2000.

[2] M.A. Bashar, G. Krishnan, M.G. Kuhn, E.H. Spafford, and Jr. S.S. Wagstaff. Low-threat security patches and tools. In *Proc. of the IEEE Int. Conf. on Software Maintenance*, pages 306–313, Bari, Italy, 1997.

[3] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 143–157, Oakland, CA, USA, 2008.

[4] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proc. of the ACM Symp. on Operating Systems Principles*, pages 298–313, Bolton Landing, NY, USA, 2003.

[5] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[6] John R. Douceur. The Sybil attack. In *Peer-to-Peer Systems: Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, volume 2429 of *Lecture Notes on Computer Science*, pages 251–260. Springer-Verlag, March 2002.

[7] Pál Erdös and Alfréd Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[8] Halvar Flake. Structural comparison of executable objects. In *Proc. of the DIMVA Conf. on Detection of Intrusions and Malware & Vulnerability Assessment*, pages 161–173, Dortmund, Germany, 2004.

[9] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, February 2002.

[10] Christos Gkantsidis, Thomas Karagiannis, Pablo Rodriguez, and Milan Vojnović. Planet scale software updates. *ACM SIGCOMM Computer Communication Review*, 36(4):423–434, 2006.

[11] Håvard Johansen, André Allavena, and Robbert van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proc. of the ACM Eurosys*, pages 3–13, Leuven, Belgium, 2006.

[12] Håvard Johansen, Dag Johansen, and Robbert van Renesse. FirePatch: Secure and time-critical dissemination of software patches. In *Proc. of the IFIP Int. Information Security Conference*, pages 373–384, Johannesburg, South-Africa, 2007.

[13] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. of the ACM Symp. on Operating Systems Principles*, pages 282–297, 2003.

[14] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[15] Dahlia Malkhi, Yishay Mansour, and Michael K. Reiter. Diffusion without false rumors: on propagating updates in a Byzantine environment. *Theoretical Computer Science*, 299(1-3):289–306, 2003.