Securing Software Update Systems in GENI

Justin Samuel and Justin Cappos

Computer Science & Engineering
University of Washington
{jsamuel,justinc}@cs.washington.edu

Abstract

Software update systems are a known source of vulnerability. The Global Environment for Network Innovations (GENI) project is highly exposed to these vulnerabilities. In order to secure GENI, it is necessary to provide a method by which both new and legacy software update systems can be protected. This document outlines the plan for the design, implementation, and deployment of a software update framework that can be used to secure software update systems on GENI.

1 Introduction

Software update systems are critical to maintaining the security of modern systems. When vulnerabilities are discovered in software installed on client systems, these security bugs must be patched in order to keep the client secure. Software update systems running on the clients have the responsibility of downloading and installing updated versions of the software in a timely manner. The ability for software update systems to function correctly, especially in the face of a malicious adversary, is therefore paramount.

Despite their crucial security role, software update systems have a significant number of attack vectors that are unprotected. These allow an attacker to revert clients to versions of software that have security flaws, deny a client updates indefinitely, or even allow the attacker to install arbitrary code [3, 4, 10]. Even worse, software update systems are ubiquitous and so provide an attack surface on every component in GENI. Furthermore, conventional defenses like IDSes and firewalls are unable to protect against these threats. As a result, an attacker with a minimal amount of technical knowledge can cause a huge amount of damage by attacking software update systems [1].

However, solving the security problems in software update systems isn't as simple as building one secure software update system and requiring that all devices on GENI use it. The vast majority of the functionality in modern software update systems goes into safely applying the software update to the application, including running application-specific pre-installation and post-installation code. This application-specific functionality tends to be different for each software update system and so attempting to replicate it in designing a single system becomes intractable given the huge number and diversity of software update systems and applications.

Without an easy-to-integrate framework for securing software update systems, GENI projects will either make no attempt at software update security or will fail to implement it properly, as has been the case with projects outside of GENI. Further, without the ability to secure legacy software update systems, some portion of GENI projects will continue to rely on existing insecure software update systems.

We will address the vulnerabilities in new and legacy software update systems used in GENI projects by designing and implementing a secure software update framework to protect these software update systems. The framework will alleviate the need for projects to choose between being insecure or creating their own—and likely incomplete—solutions to these pervasive vulnerabilities. During the stages of the framework's development as well as after its completion, we will integrate our framework with different software update systems used within GENI in order to vet its usability. We will also provide assistance to projects performing their own integration of the framework with their software update systems.

The remainder of this paper is organized as follows: Section 2 provides general background on the vulnerabilities of software update systems and the reasons GENI is at particular risk. In Section 3 we present the planned architecture of our software update framework. Section 4 discusses the integration of this framework with GENI and Section 5 concludes.

2 Background

There are two major types of software update systems: package managers and application updaters. Package managers are software used to install and update multiple, separate applications on a system. Application updaters are different in that they only perform the role of updating a single application or set of closely-related ap-

plications. Both are responsible for securely updating installed software to newer versions released by developers who maintain the software. These updates can include new features but they can also be urgent updates that fix critical security problems which have been discovered in the software.

Major vulnerabilities in software update systems, however, threaten the security of any system that is reliant on them. This threat is more than theoretical; there already exist frameworks for exploiting common vulnerabilities in software update systems [1]. These vulnerabilities are the result of problems ranging from implementation bugs to fundamental design flaws. Implementation bugs found in software update systems include replay attacks, where an attacker can trick a software updater into installing old versions of software with known vulnerabilities. In addition to other common implementation bugs, software update systems suffer from crucial design flaws, as well [3, 4, 10].

Many software update systems have a complete lack of security, while others have serious design flaws such as reliance on the security of a single, online key as well as lack of selective trust delegation [6]. By having security rely solely on the download of files from the update repository over HTTPS, these systems have a very weak single point of failure through which all clients can be compromised. The most obvious failure case with this model is when a repository that provides the software updates is compromised, resulting in all clients being subject to installation of arbitrary, malicious software. Even in cases where security comes from cryptographic signatures on software or metadata, existing software update systems suffer complete loss of security in the event of a single key compromise, a problem which has previously threatened millions of systems worldwide [5, 9].

2.1 Vulnerability of GENI

The GENI infrastructure is at risk from software update system insecurity for several reasons. These include risks common to any testbed as well as risks to which GENI is more susceptible because of its unique resources and flexibility.

Programmable networking components make man-in-the-middle attacks easier. A large part of what sets GENI apart from testbeds like PlanetLab is the control over the center of the network. However, this programmability increases the likelihood that an attacker may control the center of the network. Our previous work [4] has shown that an attacker that can launch a man-in-the-middle attack can compromise most software update systems. Since GENI is so widely programmable, software update systems on GENI are much more vulnerable than on other networks.

GENI runs diverse end host software. It seems likely that researchers will have the flexibility to load an operating system of their choice onto some subset of GENI end hosts (possibly using virtualization). As a result, GENI will run a much broader set of software update systems than other networks. This means that GENI will need to worry about security holes in many more software update systems than other networks.

Service composition increases vulnerability to attack. GENI is likely to have services that provide improved functionality to clients. However, the compromise of a service may provide the attacker with the ability to compromise a large number of clients. For example, a compromised data transfer service provides an attacker with man-in-the-middle capabilities, while the compromise of a software provisioning service may give an attacker full control over all of the managed clients.

GENI will have a high rate of software updates. Resources on GENI are likely to be configured and reconfigured on a repeated basis (for example, as a different researcher uses the resources). In many cases, these reconfigurations will be the direct result of a software update system. Since the rate of software updates is higher, the potential for attack is greater.

The resources on GENI are attractive to attackers. The GENI facility has a vision of providing researchers access to a wide variety of advanced hardware. The GENI infrastructure is highly desirable because it will allow researchers to conduct experiments and test protocols at a scale, speed, and level of control not supported by current networks and testbeds. However, these characteristics also make GENI attractive to malicious parties. Unauthorized parties may try to use GENI's infrastructure as a high bandwidth way to launch DDOS attacks, as a distributed malware hosting service, or even as a platform for cyber warfare.

3 Architecture

Our design provides a universal software update framework and corresponding set of libraries that are only responsible for the aspects of software update systems that impact security. Other aspects of software update systems, such as pre-installation and post-installation tasks, remain the responsibility of the specific package manager or application updater that uses our framework. It is because of this wide variety of case-specific needs that it is important to provide a general framework to secure software update systems rather than a software update system implementation that is usable by only a few projects.

In order to provide a general security layer for software update systems, our framework provides three major components: a developer push mechanism, a repository library, and a client library.

The role of the developer push mechanism is to take a software update that is ready for release and create security metadata for it that is signed by the developer's key. Both the software and signed security metadata are then uploaded to the repository using the developer's credentials. This developer-signed metadata will ultimately be used by the client library when determining trust in individual updates that are available for download.

The repository library receives software updates from developers and is responsible for making these software updates available to clients. This process involves maintaining a list of software updates for this application and the associated developer and software update metadata. The repository keeps an online private key [4] that it uses to attest to the freshness of the data that is served. However, this key is rotated frequently and is not otherwise trusted, thus a compromise of the key only allows freeze attacks for the period of validity of the key. Our use of selective trust delegation allows the client library to reason about whether a developer should be trusted rather than forcing the client to trust the repository to make this determination.

The client library performs several actions. First, the client library must retrieve updates from the repository library without compromising security (for example, it must prevent endless data and slow retrieval attacks [6]). Next, the client must reason about the security metadata it retrieved in order to decide which software updates should be downloaded. Finally, the client must download the various files of which the software updates consist and make them available to the software update system for installation.

3.1 Application to New Systems

As software update systems are developed for new applications on GENI, these can be secured using the components described above. New software update systems are written such that they makes calls to the client library in order to check for the existence of updates and obtain files that are part of an update. Upon successful return from these calls to the client library, a software update system can perform any required updates using the obtained files without needing to perform further security checks on those files. Security has been guaranteed by our framework's client library.

If our client library detects problems with the updates or any part of the process of obtaining the updates, it raises an exception. The software update system using the client library can handle this exception in a manner appropriate for the situation. For example, the software update system may log the failure, notify an administrator by email, notify a user through a graphical interface, etc. The action to take in the event of detection of an anomaly cannot be generalized by our framework for all software update systems because the appropriate action to take will vary widely.

In order to release a new version of software that will be successfully retrieved by the client library, developers use the developer push mechanism provided by our framework. On the software project's repository, the repository library then generates new metadata which covers the latest release pushed by the developer. At this point, the release is available to clients.

3.2 Application to Legacy Systems

In order to secure software update systems across all of GENI, it is not only important to secure newly developed software update systems but also to secure existing software update systems. To secure these legacy systems, two additional components are used. The first is a legacy interception library that intercepts traffic from a legacy software update system and translates it into calls to the client library. The second is a tool that simplifies the process of pushing software updates that originate from a legacy repository.

In essence, legacy systems are protected using the same client library, repository library, and developer push mechanisms as are new systems. The difference is that instead of modifications being made to the legacy software update systems, all communication they make with repositories is intercepted and passed through the client library. As the client library requires specific metadata generated by the developer push mechanism and repository library in order to ensure security (metadata which the legacy repository does not have), the release maintainer uses the legacy repository retrieval tool in conjunction with the developer push mechanism to simplify pushing software that originates from a legacy repository. This legacy repository retrieval tool must understand the metadata format of any legacy repository that it supports.

Throughout the actual update process, the legacy software update system remains unaware of the security being added to its update process. If the client library detects security violations when retrieving updates, the legacy interception layer causes the legacy update system to see what appears to be a generic TCP connection failure. Note that if the legacy update system uses HTTPS and correctly validates SSL certificates, either its update URL or its trusted certificates must be modified in order to secure the legacy system in this way. This allows all software update systems to benefit from the security provided by our framework's client library.

4 Integration with GENI

The development of the secure software update framework will consist of three distinct, one-year stages. Each stage will enable integration with a larger number of

projects. In the first year, the focus will be on creating a cross-platform software update framework that secures the process of checking for, retrieving, and authenticating updates. At this point, the framework will protect software update systems from vulnerabilities including replay and freeze attacks that can be perpetrated by a man-in-the-middle attacker. In the second year, the software update framework will support key management and selective trust delegation. These are essential aspects of the long-term security of any software update system. In the third year, we will implement in the framework the tools necessary to secure legacy software update systems.

In order to obtain early integration experience as well as to allow GENI projects to benefit from increased security as soon as possible, we will perform incremental integration with other projects. It is important to note that the specifics of our integration plan will need to adapt to both the evolving nature of many GENI projects as well as the impact of design decisions in the software update framework that will be made later in the framework's development.

The first GENI projects to be integrated with the secure update framework will be Million Node GENI [7] and Raven [8]. Integration with these projects will begin after completion of the first stage of the software update framework's implementation. The Million Node GENI project develops an application that runs on many end-user systems across multiple platforms, including Windows XP and Vista, Mac OS X, and Linux. The Million Node GENI project has written a standalone application updater which is installed alongside their application on end-user systems. This application updater has the responsibility of securely obtaining and installing updates. Though this application updater does protect against some known attacks, it is vulnerable to endless data attacks and slow-retrieval attacks. Integration with the first stage of the software update framework will provide protection against these attacks.

The Raven project provides a suite of tools that include both a package manager as well as utilities for administrators to collectively manage their systems. Administrators have the ability to install, upgrade, and remove arbitrary software from systems using Raven. Though Raven is based off of Stork, a package manager which was designed for security, it is vulnerable to known threats posed by man-in-the-middle attackers. Raven's security will therefore benefit from integration with the first stage of our software update framework.

The second year of our framework's development will provide selective trust delegation as well as key management capabilities. The proper design and implementation of this functionality in software update systems is essential for their long-term security. These are challenging problems that we have observed security experts have had a hard time getting right [11]. The security ramifications of various trust delegation models and key management approaches will be considered carefully. We will solicit a great amount of feedback during this stage before implementing a solution. Once implemented, we also hope to integrate these changes with Million Node GENI as well as working with the Tor project [12] to integrate our framework with their application updater [13].

In the third and final year of our framework's development, we will extend its security mechanisms to support legacy systems. This will include the ability to interpose on the insecure communication with update repositories that is performed by many existing software update system. The specific GENI projects we integrate with at this time will largely depend on which legacy software update systems are employed within GENI at that time. This will likely include existing software update systems with known security issues [14, 2].

5 Conclusion

The vulnerabilities in software update systems, though pervasive, can be addressed. Leaving the task of protecting against these vulnerabilities to each implementation of a software update system, however, leaves many projects faced with a choice between diverting their already limited resources toward addressing these problems or focusing on innovation with insecure systems. By protecting both new and legacy software update systems used within GENI from known and exploitable vulnerabilities, this project assures that GENI will remain secure. This is helpful not only to the individual projects but also to the entirety of GENI, both in terms of security as well as reduced potential for negative publicity from compromises. The software update framework allows this increased security to be achieved without burdening developers.

References

[1] Francisco Amato. Isr-evilgrade. `http://www.infobyte.com.ar/down/isr-evilgrade-Readme.txt`.

[2] Apt - Debian Wiki. `http://wiki.debian.org/Apt`.

[3] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure Software Updates: Disappointments and New Challenges. In *1st USENIX Workshop on Hot Topics in Security*, pages 37–43, Vancouver, Canada, Jul 2006.

[4] Justin Cappos, Justin Samuel, Scott Baker, and John Hartman. A Look in the Mirror: Attacks on Package Managers. In *Proc. 15th ACM Conference*

on Computer and Communications Security, pages 565–574, New York, NY, USA, 2008. ACM.

[5] Paul W. Frields. Infrastructure report, 2008-08-22 UTC 1200, Aug 2008. https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html.

[6] S. Baker J. Cappos, J. Samuel and J. Hartman. Package Management Security. Technical Report 08-02, Department of Computer Science, University of Arizona, 2008.

[7] MillionNodeGENI - GENI: geni - Trac. http://groups.geni.net/geni/wiki/MillionNodeGENI.

[8] ProvisioningService - GENI: geni - Trac. http://groups.geni.net/geni/wiki/ProvisioningService.

[9] Critical: openssh security update, Aug 2008. http://rhn.redhat.com/errata/RHSA-2008-0855.html.

[10] Justin Samuel and Justin Cappos. Package Managers Still Vulnerable: How To Protect Your Systems. *;login: Magazine*, Feb 2009.

[11] Thandy attacks / suggestions. http://archives.seul.org/or/dev/Dec-2008/msg00010.html.

[12] Tor: anonymity online. http://www.torproject.org/.

[13] Thandy: Automatic updates for Tor bundles. https://git.torproject.org/checkout/thandy/master/specs/thandy-spec.txt.

[14] Yum: Yellow Dog Updater Modified. http://linux.duke.edu/projects/yum/.