GENI Facility Security

GDD-06-23

GENI: Global Environment for Network Innovations

Distributed Services Working Group

September 15, 2006

Status: Draft work in progress (Version 0.5)

This draft is a rapidly evolving work in progress, being made public in order to solicit feedback; comments are welcome and should be directed to the authors: tom@cs.washington.edu and reiter@cs.cmu.edu . Due to the active editing process, some portions of the document may be logically inconsistent with related documents being produced by other GENI working groups.

Authors:

Thomas Anderson, *University of Washington*

Michael K. Reiter, *Carnegie Mellon University*

We would like to thank the GENI Distributed Services Working Group members, Guru Parulkar, Wade Trappe, and Alan Karp for helpful comments that contributed to this document.

Table of Contents

1 Introduction

The GENI facility promises to enable research of a scale that has previously not been possible. Experiments will be run with a worldwide geographic span, with access to compute, storage and networking resources that are unmatched in any experimental infrastructure available today. This is the source of its promise as a tool for enabling fundamentally new types of research in large-scale networks and distributed systems.

At the same time, however, the resources it embodies would be a formidable attack platform in the wrong hands. As a fully programmable substrate, a GENI slice can be configured to perform a wide range of undesirable actions against both Internet targets and other GENI resources. Even worse to contemplate is an attacker commandeering GENI components outside the confines of a slice, i.e., with greater control over GENI components than a slice should be granted. It is not difficult to envision numerous possible malfeasant uses of GENI; we need look no further than today's so called "bot-nets" and the myriad applications they support. Bot-nets today are used to propagate malware, launch denial-of-service attacks, forward spam, and traffic illegal content, for example. Without proper precautions, GENI will offer to ill-intentioned users a programmable infrastructure not unlike bot-nets today—only much better provisioned and easier to use.

PlanetLab, the testbed that has most directly influenced GENI, has already weathered such misuses, and so perhaps we should conclude that the methods used to deal with such misuses in PlanetLab are suitable for GENI. However, we believe that four factors require that GENI incorporate more advanced defenses.

- Because GENI will embody resources that far exceed those available in PlanetLab, the potential that GENI poses for a devastating attack against the Internet is far greater than that posed by PlanetLab. Put simply, whereas PlanetLab is a pistol, GENI will be a canon. As such, PlanetLab's largely manual and reactive procedures for addressing misuse might be insufficient, due to the damage that GENI could inflict at the hands of an attacker before the attack is confined.
- GENI will contain a highly diverse set of hardware resources, providing programmability across every layer of the network stack from outdoor sensor and wireless nodes to reconfigurable router and optic hardware. What works in one context may not work as well in another. Thus, we need to take a fresh look at vulnerabilities and proposed solutions.
- If GENI realizes its full promise, then its sheer scale of usage could result in a stream of incidents and complaints that would outpace the human-intensive procedures used in PlanetLab to manage them. This is a problem all-too-familiar to operators of large networks today, and could be magnified in GENI owing to the advanced types of research conducted on the facility. We thus believe that it is prudent to integrate technological methods to stem this tide now, rather than to attempt to choke it off once the flood has begun.
- As an infrastructure being constructed at the behest of the National Science Foundation and at considerable expense to the public, GENI will (and should) come under greater scrutiny from many quadrants than PlanetLab ever has. The public relations consequences resulting from constructing, in the worst case, the world's largest and most capable attack

infrastructure could be felt by the NSF and the computer science and engineering communities for years to come.

Despite the above challenges, we believe that technical and operational solutions exist that can lead to an effective security architecture for GENI. Our goal is not to eliminate the possibility of a successful attack – no computer system connected to a global network has ever reached that bar. Rather, our goal is to use technical and operational means to prevent, detect, and manage attacks, so as to render the GENI system to be both safe and usable by its target community of experimental network and distributed systems researchers. A further goal is to illustrate "security done right": to show how security should be architected into globally distributed systems of networks and computers.

A guiding principle is that we believe that GENI should be designed to operate in a "do no harm" posture. This level of accountability is far beyond the capability of the Internet, and it implies several things. First, an experiment should be given only the specific permissions that are needed for it to run, and are merited by its prior validation steps. There is no need to allow every student running an experiment on GENI the ability to flood the Internet with unwanted packets; protecting the Internet and the public wireless spectrum from runaway experiments should be a requirement. Rather, such privileges should be granted and enforced in a measured fashion as an experiment is incrementally validated and shown to do no harm. Obviously, we mean "harm" in only the broadest sense. As any researcher or even any company deploying a new Internet service is acutely aware, it is sometimes impossible to completely prevent all possible complaints, no matter how careful the validation. The Internet today is a hodgepodge of unpublished and often unpredictable filter rules, so some issues with respect to the inadvertent consequences of experiments will come up from time to time, requiring the ability to trace the cause of the complaint back to the responsible experiment (and experimenter), so that the problem can be addressed.

Again in contrast with the Internet, the facility as a whole should continue to function even while remedying most types of accidental or intentional misuse. For example, we envision a "kill switch" that will enable an operator to quickly and completely suspend a misbehaving experiment or component, while allowing the rest of the system to proceed. However, if GENI enters a period where activities of some components or slices cannot be adequately monitored or controlled to ensure their safety, then GENI should restrict those activities by other means to a point where safety can be assured. For example, in the face of a denial-of-service attack on control channels to and from GENI Management Central—say, rendering certain components invisible to the control and monitoring infrastructure—GENI should retreat to a state that prevents those components from communicating with the Internet and, potentially, other critical GENI resources. Taken to the limit, this implies that GENI should shut itself down in certain circumstances where "off" is the only state where safety can be assured. Obviously the circumstances requiring this should be few, lest GENI's scale, widespread deployment, and visibility make GENI an inviting target for "denial of GENI" attacks.

Another consequence of the above is that GENI itself should resist low-level compromise to the extent that the state-of-the-practice permits. We anticipate that many GENI nodes will be connected to the legacy internet—e.g., the initial GENI management plane is expected to run over the legacy internet—and as such they will need to be defended from compromise as any internet-connected host must be. It is not the goal of this document to cover such defenses in

detail (instead, see [4] for a more complete discussion), but the regimen is familiar: use off-the-shelf operating systems and system software and apply security updates promptly and completely across the entire system; provide only the minimum number of open ports, and those requiring authenticated access (of course, specific sandboxed experiments may open additional ports); actively monitor the system to detect and report intrusions (e.g., attempts to modify the boot file system); etc. While we touch on these issues, our primary focus here is the uncommon challenges that the GENI facility introduces and the mechanisms that we recommend be developed to address those challenges.

Following standard industry practice, we believe that the GENI operations and governance teams should practice advance "test runs" of their procedures to handle serious security breaches. No technical solution will be 100% foolproof in preventing attacks, especially given the amount of pre-existing (and vulnerable) software that GENI needs to use in order to be cost-effective. We of course hope that GENI will help foster research that will lead to a more secure future Internet, but until that time, the cost of an attack is related to how quickly it can be addressed—something that can be optimized with advance planning and practice.

2 Threat Model

There are three broad classes of attacks that must be addressed by the GENI security architecture and by its operational procedures. First, attacks may be launched by outsiders on the GENI infrastructure, either as a denial-of-service attack, or simply to gain control of GENI resources. Second, and related, we need to prevent and limit the impact of accidentally or maliciously misbehaving GENI experiments on the outside world; similarly, we must limit the impact of attackers posing as legitimate GENI experimenters. Finally, we need a level of isolation between experiments, so that GENI cannot be surreptitiously used by one experimenter to disrupt another.

We discuss these three types of attacks in this section by providing a list of specific threats that the GENI security architecture must address. The threats are listed according to our estimate as to the relative frequency of that particular type of problem; for example, accidentally misbehaving experiments are likely to be a somewhat frequent occurrence on a platform designed to support experimental investigation, while determined attacks against the GENI software are relatively less likely, but more serious. Fortunately, many of the same technical solutions can be applied to both root causes. Note that the threats we list below are not intended to be completely mutually exclusive: systematic attacks against GENI may combine multiple elements, and thus the facility needs to be able to deal with all of these types of problems simultaneously. We note as well that despite the Internet being explicitly designed to support experimentation and evolution in its protocols, the Internet's architecture is poorly suited to deal with any of these types of problems.

- Runaway experiments that cause unwanted Internet or RF traffic. Experience with PlanetLab suggests that unintentional misbehaving experimental code will be a common occurrence on GENI. We believe a process is needed to assign and enforce specific, minimal privileges appropriate to each experiment, e.g., so that a novice user's mistake does not have global consequences. Another requirement is a rapid "kill switch" to enable operations staff to quickly suspend a misbehaving experiment. A companion document discusses external

RF monitoring to detect and stop experiments that inadvertently pollute the RF spectrum [33].

- Runaway experiments that disrupt the execution environment for other experiments within GENI, e.g., by exhausting disk space or file descriptors. These issues can be handled by providing stronger isolation between experiments and by monitoring shared resources for unexpected usage patterns.
- Experiments that escape their isolation boundaries and accidentally (or if an experiment is taken over by an attacker, maliciously disrupt) the networks of hosting organizations (e.g., see [29]). The GENI facility must ensure that hosting organizations are not put at significant risk for contributing resources to GENI, and the GENI effort must take measures to convince hosting organizations that problems are rare and dealt with promptly.
- Legitimate experiments that trigger true or false alarms in other parts of the Internet or wireless spectrum. This requires the ability to trace the source of packets or signals back to the responsible party, so that the problem can be fixed and prevented from recurring.
- Misuse of an experimental service by an end user. For example, one example experimental service conceived for GENI is to run a virtual ISP supporting a novel internal architecture. Such an experimental ISP might be used by a malicious user to launder illegal packets. We expect this set of concerns to be addressed by establishing GENI-wide standards for experiments offering packet delivery services (or their equivalent) to end users. For example, GENI might require that an experimental ISP provide basic monitoring or tracing tools for legitimate law enforcement enquiries, and indeed we believe GENI should provide a baseline toolkit for use by experimenters in meeting this requirement. A companion document describes experimenter support in more detail [2].
- Theft of an experimenter's credentials to use GENI. Unfortunately, it is well-known within the security community that users are often careless with the keys used for authentication, if only because key compromises are silent until it is too late. Carefully calibrating privileges to match the experimenter's sophistication is one avenue (e.g., users likely to be careless with their keys would be given more limited privileges); another is to use technical means described below (Section 5) to make it more difficult for attackers to gain access to user keys.
- Corruption of the host operating system software running on the experimenter's desktop machine. Since end host corruptions are endemic on the Internet today, we need to make it easy for the GENI operations staff to revoke and replace end user keys and privileges after such break-ins. Even so, this is perhaps the most likely avenue for malicious attacks against GENI.
- Corruption of the systems software running on one or more components. An attacker might gain temporary control over a node by first gaining access to a GENI account (e.g., by corrupting an experimenter's host computer), and then launching an "experiment" that exploits a vulnerability in the node operating system software to gain control over the node. One important step is to actively manage all GENI hardware, e.g., to proactively keep all operating system software up to date with known security patches. This means that any changes we make to host software be minimal, so that patches can be applied quickly. Another important step is that components should be configured with the minimal number

of open ports. Yet another important step is to discover problems quickly, e.g., by providing for continuous monitoring of anomalous node behavior by GENI operations. (This is of course made more complicated by the fact that the experimental architectures and services running on top of GENI are by their very nature, anomalous!) Yet another important step is to be able to fix the problem quickly once it is discovered. The emergence of trusted computing hardware [28] and the integrity measurement architectures it supports (e.g., [30]) should provide a mechanism for GENI operations staff to reset every node in GENI to a known, good state.

- Denial of service attacks against the GENI management infrastructure. As we mentioned above, GENI should fail "off" to avoid providing an avenue for an attacker to take control, and then use denial of service to prevent the operations staff from taking countermeasures. Technically, this can be accomplished by requiring privileges to be frequently refreshed. Initially, we envision GENI management commands will be carried over the Internet for convenience (subjecting them to all the problems of Internet security and reliability); as GENI construction proceeds, however, it may be possible to move the GENI control plane to running over GENI itself, reducing the likelihood of such attacks (see Section 7).
- Direct attacks against vulnerabilities in the GENI management software. GENI is a complex distributed system, and therefore special care must be taken to avoid vulnerabilities in its implementation. One step is the explicit modeling of trust relationships between GENI components as described below. Another important step is to observe that the software development processes adopted for GENI software are critical to the security of the GENI facility. It is well-known that poor software quality is the source of numerous types of serious security vulnerabilities in practice (e.g., buffer overflows and format-string vulnerabilities). We believe it is imperative that GENI software development processes be adopted so as to eliminate, to the extent practicable, these types of vulnerabilities. While specifying software development processes is outside the scope of this document, an example might be that all GENI-defined interfaces and protocols be adopted only after an open, public review of potential security vulnerabilities, that changes to interfaces be made only through a similar formal process, and that conformance tests be generated (ideally, automatically) from a formal specification of the interface. Where practical all GENI software should be implemented to be type-safe, e.g., via tools such as CCured or languages such as Java. Where type-safety is impractical, e.g., in modifications to an existing operating system implemented in C, standard practices such as software verification tools and test suites can be used to reduce the likelihood of vulnerabilities. We also believe that serious consideration should be given to requiring that source code produced for GENI be made public, so as to allow for independent security analysis. However, we do not believe it is a cost-efficient use of GENI resources to require every aspect of the management software to be robust to arbitrary malicious attacks by privileged insiders (so-called Byzantine attacks). Rather, we intend to rely on detection, confinement and resetting to a known good state to correct intrusions when they occur.

Two additional issues that we do not yet consider in depth are the privacy of experimental data and the privacy of management policy. Preventing unauthorized access to information stored in GENI can be accomplished using the flexible access control architecture described later in the document. However, preventing all forms of information leakage while an

experiment is running is an open research challenge, and one for which we hope GENI will help facilitate the development of technical solution.

3 Security Requirements

This list of vulnerabilities suggests several requirements for GENI's security architecture:

- Explicit Trust: Privileges in a distributed system should be managed explicitly and formally. Enforcing security in GENI is something of moving target, as the facility will be used during its construction, and progress from a single management entity to a more federated model. Thus we need a security model that can evolve along with GENI. The access control approach we describe below (Section 4) is intended to provide the required flexibility, rather than hard-coding trust relationships. One concrete benefit of this approach is late binding—that we can start with a PlanetLab-style aggregation and delegation of rights, and add federation later on, without needing to know exactly how trust will be managed in the federated system. Without explicit trust, it is likely that trust will be unintentionally misplaced, leading to system-wide vulnerabilities that can be exploited by the wily attacker.
- Least Privilege: The principle of least privilege is a tenet of computer security that requires each component of a system be given exactly the authority it needs to perform its tasks and no more. Failures to implement this principle are ubiquitous, and we face the consequences frequently. For example, most web servers do not need to be able to open connections to arbitrary addresses in order to perform their tasks. Yet this is permitted, and exactly this ability has been used numerous times in the epidemic spread of worms. While achieving least privilege in an absolute sense is arguably not feasible, it is our belief that the GENI facility should approach least privilege as far as is practicable. Least privilege can secure the GENI facility from malicious software, accidental violations, or just simply resource exhaustions—in general, it can mitigate the risks caused by runaway experiments. It is also equally useful in securing the experimenter's environment against attacks from other experiments or faulty system software.
- Revocation: Despite our best efforts, it is inevitable that keys, slices, and systems will be compromised in GENI. Thus a critical requirement for GENI is to be able to quickly revoke and replace keys, suspend all permissions (e.g., slices) derived from a compromised key, and (as in PlanetLab) reset each node to a known secure state.
- Auditability: The possibility of compromise also requires us to be able to trace why a problem occurred so that it can be prevented from recurring. As we describe below in Section 6.2, PlanetLab contains an initial implementation of some of the needed functionality: it logs every packet header sent by every slice running on PlanetLab. Our intent is to go well beyond what PlanetLab provides; we need to know which slice is responsible for each packet, but also we need to be able to determine the entire chain of responsibility (from central administrator to local administrator to local user) that gave the user a specific capability that was misused.
- Scalability: With large-scale distributed systems, simple schemes such as using a small set of authentication servers and/or replicating information required by authentication and authorization tasks are not feasible. For instance, it is now commonly accepted in the grid

computing community that the grid security infrastructure needs to move beyond the use of grid-wide unique IDs and a global table mechanism (GridMap) for replicating authorization data. PlanetLab also is limited by its reliance on a global table mechanism. We propose a specific more scalable authorization architecture below.

- Autonomy: A key requirement for GENI is the ability to federate autonomous facilities. A GENI site should be able to authenticate and authorize requests from users in other sites, support delegation of rights, and it should be able to do so without requiring centralized trust. While it might be possible to extend the PlanetLab model in an ad hoc way to support autonomy, below we advocate a more principled approach.

- Usability: The user must be explicitly modeled as part of the security architecture. Any system that is hard to use will be evaded and ignored. The implication is that we need to make it easy, rather than heavyweight as in PlanetLab, for users to create roles, restrict rights, etc. We also need to make it easy for users to protect their private keys. In essence, secure system and user behavior must happen by default.

- Performance: As with usability, the performance overhead of providing security needs to be modest, or users will have an incentive to disable or evade the system. In practice, this means managing security information (such as certificates delegating rights to a specific set of users) as cache-coherent, distributed state. Caching means that lookups can be local and fast in the common case, without compromising system semantics.

The rest of this document provides a partial list of technologies we propose to use in achieving these requirements. We emphasize this is intended to be a partial list, focusing on those aspects of GENI that present uncommon challenges and that we recommend be addressed via new development. Given this restricted focus, there are several notable but intentional omissions from this document:

- GENI nodes' operating systems and the isolation properties they enforce. While this is fundamental to GENI facility security and numerous other aspects of GENI function, we expect that the initial GENI deployment will utilize the best available operating systems at that time, as it is outside the scope of GENI to construct new secure operating systems for the range of devices that GENI will incorporate.

- Defense of GENI nodes from compromise via attacks from the legacy internet. Those GENI nodes that are connected to the legacy internet—and we expect that at least at first, most or all will be for management purposes—will be at risk of compromise from the outside. We expect this threat to be countered as it must be for any internet-connected node (e.g., see [4]), i.e., with an appropriately configured firewall, elimination of unnecessary services, prompt application of patches to necessary services, traditional intrusion-detection, and so forth. In fact, protecting a dedicated GENI node from compromise from the legacy internet might be simplified by its role: e.g., the node's firewall can drop connections to ports not associated with experiments from all but recognized GENI Management Central computers. Of course, experiments in slices may still be subject to compromise if they communicate with the legacy internet, but the protections we describe below will confine this threat just as they would for any wayward experiment.

A related way in which we restrict our focus for the remainder of this document is that we elide treatment of sensor networks and other special-purpose "edge" systems that present resource or connectivity constraints atypical of a general-purpose computing environment. Due to the resource constraints of these systems, the security risks associated with them are primarily inward-facing—i.e., that these components will be disrupted in a way so as to hinder experiments on them or steal information from them—but their compromise poses relatively little direct threat to the systems to which they connect. As with the rest of GENI, best practices should be applied to defend them (e.g., see [33] for a discussion), but since we focus our attention on new developments needed to meet the unique security threats posed by GENI, we do not cover those other elements here. Nevertheless, several of the mechanisms proposed here could play a role in those edge systems, at least for those elements that offer adequate resources to execute them.

4 Access Control

The core of our proposed security architecture for GENI is a pervasive and unified access-control infrastructure. In security parlance, *access control* refers to means to reach a yes-no decision as to whether a requested access should be granted. The decision is reached by a resource monitor, based on evidence as to whether the requested access conforms to security policy. The goal of the architecture we propose is to provide a unified and yet flexible mechanism for resource monitors to reach these decisions.

Access control is often intimately tied to authentication, and so a side-effect of the architecture we propose is the provisioning and operation of a distributed Public Key Infrastructure (PKI) and Certificate Authority to allow strong identities for facility users. Although PKIs are hard to bootstrap, we note that GENI has a natural advantage that we believe can pave the way towards more widespread use of PKIs: every site has a local administrator who can establish and vouch for the credentials for each specific GENI user and physical device. Authentication is required for both the network facility itself, to grant access to applications and services and provide a basis for resource isolation, but also for applications and users. A flexible and accessible public-key or other authentication service, along with the software and resources to manage it, will bootstrap both GENI itself, and the development of applications on top of it. This service must include the development of libraries to allow a variety of applications to use the service and the development of guidelines for how and when applications should use the service.

4.1 Background

One way of understanding our approach to access control is by analogy to the access control framework already found in Java. Java implements a *security manager* that can be invoked from any point in a Java program, with an action that has been requested and a security policy that must be checked in order for that access to be permitted. The security manager implements a systematic procedure for determining whether the access complies with the provided policy; if not, it raises a security exception. Our goal in this document is to set out the requirements for the analog of a security manager for GENI, i.e., a systematic procedure for determining whether a particular access is consistent with a particular policy, without specifying for what accesses such checks will be performed or what the policies should be. Just as the security manager is

equally useful in Java for both system-level access checks and checks by applications that were not anticipated when the security manager was built, we would like the access control framework that we describe here to be useful to both protect the GENI instrument and to enable applications built over that instrument to protect themselves.

That said, as we will see, the analogy to the Java security manager ends there; the mechanism we advocate here has little else in common with Java's security manager. An authorization service for GENI needs to address the large, distributed nature of the target platform, and the need for designing a flexible system that can express a rich set of security policies.

In terms of existing technologies that address some of requirements described above, GENI's predecessors in both the distributed systems community (PlanetLab) and the Grid community (Globus) have primitive security architectures (key-based access control lists) that address authentication more than authorization. These systems simply distribute the public keys of users, use the keys to authenticate requests, and give execution privileges to authenticated users; there is little to no differentiation regarding the privileges of authenticated users, nor is there any mechanism for users to execute programs with restricted rights, to grant subsets of their rights to other users, or to authenticate previously unknown users. Public-key infrastructures such as X.509 [18] provide some of what is needed—they allow local sites to authenticate users even if the necessary keys are not available locally—but they are limited by the lack of support for local authorization policies.

4.2 Access-control systems and logics

Any system that implements access control does so through some type of program logic. Usually this involves checking whether the requesting party is on an *access control list*, but it might additionally involve checking whether that party is a member of a group, for example. In distributed systems, access-control decisions often must be based on policies (e.g., expressing delegations of authority, group memberships, and so forth) of different principals in the system. It is typical for these policies to be encoded in digitally signed credentials that must be assembled and presented to the resource monitor for evaluation. Numerous such systems and standards have been developed in the research community (e.g., [36][10][11][14][9][19][8]).

Since the early 90s, efforts to gain assurance in decentralized access-control systems involved modeling access-control policy and the system enforcement in a formal logic (e.g., [1][17][19][21]), so that claims about it could be made precise and verified. More recently, formal logics have been explored as a means to *implement* the access-control decision procedure (e.g., [1][35]), and have been used in such a capacity in a handful of research systems (e.g., [6]). This increases assurance further by minimizing the gap between the logic (about which results are proved) and the system implementation.

While detailing an access-control logic is outside the scope of this document, it is worthwhile to summarize how such a logic is used. In such a system, formulas of the logic are instantiated from digitally signed credentials. An example of such a credential might be a traditional certificate issued by a certification authority, but more generally the credentials can utilize richer constructs in the logic, such as groups or roles. The inference rules of the logic are then applied to these formulas to construct a proof of a required access-control policy. The required

access-control policy can be different per resource being accessed, and its formal statement will typically involve a nonce identifier so that the resulting proof cannot be replayed.

4.3 Authorization Architecture

We now provide a high-level view of our proposed system architecture and how the different components work together to provide authorized access.

Principals, including users, administrators, and machines, can generate requests or make assertions regarding privileges associated with other principals. Objects are resources, such as CPUs, files, and network devices, which are to be guarded against unauthorized access. Each object is associated with it a resource monitor that checks whether or not to grant access to the object.

We decompose the act of gaining access to a resource into two distinct steps: constructing a proof (typically, a set of certificates) that the access complies with access-control policy, and checking that the proof is valid.

The task of constructing the proof could be accomplished by any of the following means. In the simplest case, the proof could be obtained using the very mechanism that was used for finding the resource. For instance, if the principal became aware of the resource through a resource allocator (such as Emulab's assign), a resource discovery tool (such as SWORD [27]), or a resource broker (such as SHARP [15]), the same underlying tool could be extended to generate the necessary certificates and provide it to the principal. In the more general case, one could use a general-purpose theorem prover that would perform distributed queries to discover a set of credentials that would constitute the proof for authorized access (e.g., [23][5]). However, as the certificates are stored in distributed repositories, the certificate discovery process might require multiple remote accesses, potentially causing performance bottlenecks. A middle-ground that is less general but potentially more efficient would be to have an application-specific rule base for discovering the credentials and assembling the proof; the rule base would then embody a set of application-specific heuristics for finding the desired set of certificates.

The task of checking the validity of the proof is performed by the resource monitor, a task which places the resource monitor at the very heart of the authorization service. When provided with a security policy, expressed as a formula of the logic, and a claimed proof (including digitally signed credentials) that a request satisfies this security policy, the resource monitor would verify the digital signatures on all certificates to ensure their validity and then verify that the claimed proof using them is indeed a valid proof of the security policy.

The resource monitor thus embodies significant design decisions, not the least of which is with respect to what logic the resource monitor verifies the proof. There appears to be a tradeoff between the expressiveness of this logic and the ease of generating proofs of access. For example, if we were to adopt the proposal of Appel and Felten [3] that advocates the use of higher-order logic, we would be favoring extensibility over efficiency of proof generation. This logic allows the expression of policies using higher-order predicates and quantifications, and so is powerful enough to encode many different authentication frameworks (such as Taos [36], SPKI/SDSI [14], and X.509 [18]). However, with such generality, the process of automatically

generating the appropriate proof becomes undecidable in the general case [3], and moreover, it is difficult to prove certain security properties of the framework. A careful study of the tradeoffs is still needed before deciding on the appropriate formal logic.

Within the above framework, we plan to employ a rich set of certificates or declarations to implement various forms of authorizations. Following previous research efforts (e.g., [36][11][14]), we envisage the use of following abstractions:

- Identity certificates that bind principals to keys. These certificates are useful for authenticating principals.
- Authorization certificates that attest to privileges associated with principals. When combined with the identity certificates, these certificates enable authorized access.
- Delegation certificates that pass on privileges (or a subset of privileges) from one principal to another. As part of the delegation certificate, the delegator can state whether the delegatee can further delegate the privileges to yet another principal in the system.
- Group membership certificates that allow an authorizing agent to group together principals and manage their privileges in a scalable and efficient manner.
- Roles that allow a principal to voluntarily restrict its privileges, thereby limiting the dangers posed by security violations.
- Revocation of previously issued certificates, as no authorization service would be complete without mechanisms for revoking privileges associated with misbehaving principals or compromised keys.

While it is beyond the scope of this document to define the full set of actions or resources for which authority should be checked in GENI, we believe the above framework is flexible enough to be used in the following contexts:

- The service should enable fine-grained controls on resource usage of user-level experiments. It should enable the execution of user-level programs in sandboxed contexts that enforce least privilege.
- The authorization service should support the access checks performed during system administrative tasks for creating, removing, or modifying information regarding sites and nodes. It should also support the process of granting, revoking, or checking roles, and authorize site-management tasks such as rebooting nodes or setting bandwidth limits.
- The service should also be useful for implementing user tasks pertaining to updating user information, initiating and controlling experiments, including operations such as adding or removing virtual machines from an experiment.

There are a number of important technical questions that remain open in the design, including:

- What form of logic should be used in the authorization service in order to be able to handle the types of actions we plan to run on GENI? Should more restrictive logic forms be used in order to enable efficient proof generation or simplicity?

- What tradeoffs exist between generality, flexibility, performance, and assurance? Can we express various security policies without loss of efficiency? Do we need mechanisms for caching the results of certificate discovery queries and authorization checks?
- A related question concerns the granularity in which access rights are to be expressed. Fine-grained access rights allow a system to implement the principle of least privilege, but potentially at the cost of increased overhead. We might need some form of dynamic bundling of rights and/or indirection to enforce the principle of least privilege without sacrificing performance.
- What secondary mechanisms are required to limit the damage caused by security compromises? For instance, revocations could be implemented using Certificate Revocation Lists, one-time certificates, and/or short validity intervals. Which of these mechanisms is appropriate for different usage patterns?

4.4 An Example

To illustrate the use of the decentralized access-control framework that we propose, in this section we sketch an example of one way in which it might be used in GENI. (We avoid use of logical formalism to the extent possible, however.) This example is based on a similar one found in Lampson [21]. Consider a computer that an organization, say CMU, wishes to contribute for use by GENI experimenters, but only in a limited fashion. For example, CMU desires that GENI experiments be permitted to execute on this computer but not to open connections to legacy Internet hosts; rather, slivers are permitted only to connect to other GENI hosts. The following steps might be taken to enforce this policy, while permitting experiments to take advantage of this node. Below, when we say that public key **`pubkey`** "delegates authority" to an entity (e.g., another public key), we mean that the corresponding private key is used to digitally sign a message delegating authority to that entity.

1. The computer, denoted **`node`**, is initialized with a hardware private key with verification key **`nodeKey`**, which is used to sign a statement delegating authority to a private key controlled by CMU (with corresponding public key **`cmuKey`**).

2. The computer is installed with the GENI operating system. A resource monitor in the operating system that intervenes in connection requests is configured to require the each connection request be accompanied by proof that **`nodeKey says connect(…)`**. Here, **`says`** is a logical constructor typical of access-control logics (e.g., see [21]); intuitively this statement indicates that the connection request is compliant with policy attributable to **`nodeKey`**.

3. The **`cmuKey`** is used to delegate authority to the GENI management system, to authorize connections from **`node`** to only GENI addresses (and not legacy Internet addresses). In this delegation, GENI is named by its public key **`geniKey`**.

4. GENI delegates authority to a principal investigator (PI) to create a sliver on **`node`**. The rights passed to this investigator, named by **`piKey`**, include creating a sliver that connects from **`node`** to GENI addresses. This delegation is encoded in a digital credential signed by **`geniKey`**.

5. The PI (via **piKey**) delegates this authority to a group, **piKey.students**, of graduate students who are developing the experiment. In addition, **piKey** creates a credential for each such student, named by his public key **studentKey**, giving that student the authority of the **piKey.students** group.

6. Upon deploying a sliver to node, a graduate student uses studentKey to delegate to that sliver (**sliverKey**) the authority to connect to GENI nodes.

7. In order to open a connection to a GENI node, the sliver must assemble a proof that **nodeKey says connect(…)**. How it does so depends on the logic being used, but intuitively it will need to utilize the credentials described above, i.e., that formed in Step 1 that delegates authority from **nodeKey** to **cmuKey**; in Step 3 to delegate authority from **cmuKey** to **geniKey**; in Step 4 to delegate authority from **geniKey** to **piKey**; in Step 5 to delegate authority from **piKey** to **piKey.students** and to grant the authority of **piKey.students** to **studentKey**; and in Step 6 to delegate authority from **studentKey** to **sliverKey**. As such, when **sliverKey says connect(…)**, it can be inferred (via this substantial chain of reasoning) that **nodeKey says connect(…)**.

Of course, since humans do not relate to others using keys, the above delegations might instead be to named persons, e.g., the professor's name instead of **piKey**. The relationship between that name and the key **piKey** is a classic "certification authority" problem that can be solved using a GENI certification authority or via some other means of registration. While we do not think it necessary to fully specify this here, we comment that this type of certification can be encompassed with an adequately rich logic, i.e., so that the certification credentials are expressed in the same logical language as the other credentials in the system.

Another extension of the above example that we anticipate being used in GENI is including integrity measurement (e.g., [30]) of the node platform in the proof process, i.e., so that a valid proof of **nodeKey says connect(…)** can be constructed only if node is executing the approved GENI operating system, for example. We have elided this from the example above, however, for simplicity.

Several subtleties in authorization are already evident in the above example, even at its high level of informality. For example, the compromise of the private key corresponding to **geniKey** might enable a sliver to use **node** to connect to non-GENI addresses, even though this key was delegated only the right to connect to GENI addresses. This could occur, in particular, if **geniKey** is used to *define* what addresses are addresses of GENI components (i.e., by digitally signing those addresses). In this case, the compromise of the corresponding private key would enable the adversary to redefine GENI addresses, to potentially include all addresses. This threat suggests, for example, that GENI might want to have different keys for delegating authority and defining the addresses owned by GENI.

4.5 Discussion

We now address potential concerns regarding the proposed design.

Is an advanced authorization service really needed for GENI? Authorization is potentially deeply embedded in every GENI interface. If the authorization framework is left unspecified, every potential application might develop its own ad-hoc mechanism, violating basic software engineering principles and potentially resulting in rigid interfaces with limited functionality that will be very difficult to change later on. The risks of rigidity and ossification could be avoided by developing and eventually agreeing on the API for an authorization service.

Is the proposed authorization scheme more complicated than it should be? We believe that the proposed authorization service is relatively straightforward to implement and easy to use for various application scenarios. Note that one of our key design decisions is to separate the proof verifier from the proof generator. Only the proof verifier needs to be part of the resource monitor, thereby making the trusted computing base small and easy to implement. The proof generator is also simple for typical usage settings, with the proof generation mechanism just making the authorization chain more explicit. In a certain sense, one could view our proposed scheme as a methodology with ``checkable preconditions'' regarding who is authorized to perform which operations. By making the authorizing certificates explicit, the principle of least privilege could be enforced.

We also believe that the generality of our proposed mechanism is essential given that the trust relationships between GENI entities is likely to evolve over time. We need an authorization framework for GENI that can be used to support both very simple policies (a la PlanetLab) and vastly more complex ones that could arise later. Similarly, it need not immediately be used to control access to all resources on day one, but the potential is there to refine the security policies over time.

5 Protecting Private Keys

An access-control architecture is only as secure as the private keys that contribute statements to it. Other private keys might play a similarly important role for GENI, e.g., as the means to decrypt sensitive data that is collected and stored on untrusted (from the user's perspective) GENI nodes.

For this reason, we believe it important to provide facilities that help a GENI user or administrator protect her long-term private key (or any other private key for which the use should be contingent on consent of a user) from misuse if it is disclosed to an adversary. There is a spectrum of protections that might be considered as described below, in order from least secure to most secure:

- Password encryption of the private key has the obvious advantage of being familiar to the majority of would-be GENI users. However, it places an undue burden on users to choose passwords that will resist an offline dictionary attack. For example, if the private key is stored in encrypted form on the desktop machine of the user, then it is vulnerable to an attacker who gains access to the key file—e.g., from a machine backup or due to misconfigured access controls on the host—and then breaks the password via an offline dictionary attack. In addition, a key that succumbs to such an attack can be used by the adversary indefinitely, at least until the components that trust the corresponding public key (e.g., to verify signatures or create encryptions) cease doing so.

- A password-enabled private key can be made more secure by sharing control over the use of the private key with a GENI component; we will call this component a *capture-protection server* (CPS) and a private key that it protects a *capture-protected key* (CPK). The CPS limits the use of a CPK to only the owner of that key—i.e., the user who can supply the password or PIN for that CPK—and optionally provides the means by which the key can be *disabled* (temporarily or permanently) even for the legitimate user or someone who can impersonate that user. While disabled, the CPK cannot be used to perform cryptographic operations. Proposals to implement such a CPS go back at least to Yaksha [16], and build from cryptographic techniques to share control over the use of a cryptographic key between two parties (c.f., [13][12][26][24]). Modern proposals (e.g., [24]) highlight simple management properties (e.g., the CPS does not require initialization per user) and better minimize trust in the CPS. In particular, the CPS need not be in the trusted computing base for secrecy of the CPK: as long as the CPK itself remains undisclosed, the compromise of the CPS does not enable misuse of the key.

  This approach retains the familiar interface of a password-protected private key, but without relying on the user to select a password that could withstand an offline dictionary attack (e.g., a 4-digit PIN would suffice, just as for an ATM). Moreover, this approach better protects the key file in the case of its inadvertent disclosure, in the sense that the key file is useless to the attacker unless he can impersonate the user to the CPS. That is, once obtaining the key file, the attacker would be forced to conduct his dictionary attack online (where it can be detected and stopped by the CPS) versus offline as with a simple password-encrypted key file. This solution remains susceptible, however, to an attack that both captures the key file and the user's password/PIN—at least until the CPS is notified and disables the key.

  There need not be only one CPS for all of GENI. GENI Central can provide a default CPS, though it should be possible to change the CPS used for a private key, e.g., to a CPS newly set up by an institution to protect its users' keys. This is in keeping with the tenet that ultimately the only authority GENI Central has is that which institutions place in it. Modern proposals support such functionality, as well [25].

- Storing the private key solely on a hardware token (e.g., smartcard) can make it significantly less likely that an adversary will gain access to a private key. This solution is appropriate for keys whose compromise could have far-reaching consequences (e.g., the keys of GENI administrators). Methods of disabling (as in the CPS description above) a key protected via a hardware token are also possible. Of course, there are a variety of hardware tokens to choose from, including some that are tamper-resistant.

While ideally the third option would be welcomed by all GENI users, we expect that this is unlikely. In particular, most GENI users will use the system only sporadically and so might be unwilling to carry a hardware token regularly. This, in turn, might increase the frequency of occasions when the researcher needs the token but either forgot to carry it or misplaced it. Hence, for the sake of usability, we do not anticipate requiring all users to utilize hardware tokens. Rather, we suggest supporting hardware tokens for those willing to use them (and requiring them for some classes of users, such as GENI administrators) and supporting one or more CPS, run by GENI Central or by institutions, for other users. Moreover, the authorization

infrastructure outlined in Section 4 can tune authorizations depending on the manner in which the key to which authority is delegated is protected.

6 Audit Trails and Intrusion Detection

As we discussed earlier, auditing is an essential part of our proposed security architecture for GENI. Auditing complements authorization (e.g., if authorization is explicit, you can more tightly verify the audit trail).

Fortunately, auditing in this environment is simplified by the lack of sharing between virtual machines running on any node in GENI. As a result, without a compromise of the virtual machine and its containment system, users can only read and modify files within their own virtual file systems. This simplifies the need to track what happens within a virtual machine—as long as the process of logging into the virtual machine checks the authorizations (and perhaps logs them), the question of what happens solely within the virtual machine is not a concern outside the node.

This approach naturally implies that the more important aspect of auditing is what is externally visible from the node. Any traffic that is sent to or received by the node is a possible source of problems, regardless of who generates the traffic. The network is also one point of shared visibility in GENI. It is unlikely that every site participating in GENI will be able to allocate enough IP addresses such that every virtual machine has its own IP address. As a result, the IP address space and the port space will have to be shared on GENI, leading to the main reason for network-related auditing: determining what project is responsible for actions that raise alarms.

6.1 Intrusion Detection

"Alarms" can arise from various sources. In PlanetLab today, these "alarms" are most often complaints from network operators on the Internet, who object to traffic being received from PlanetLab machines (perhaps due to the alarms from their own intrusion detection systems). Relying on alarms from network operators elsewhere should not be the preferred method of intrusion detection in GENI, however. Some activities might not be detected or understood quickly, and so it is incumbent upon the designers of GENI to build the facility in such a way that GENI monitors itself for trouble-causing behavior.

Ideally, it would be possible to specify for GENI what kinds of network traffic or other activities should be detected as dangerous. Specifying and detecting such undesirable behaviors is the domain of *signature-based intrusion detection* (or sometimes *misuse detection*). Though widely used in practice today, this approach provides neither completeness, nor accuracy. As to the latter, PlanetLab experiments are notorious for raising false alarms with signature-based network intrusion detectors, due to the widely varying behaviors that PlanetLab experiments exhibit. We expect that the even wider variety of networking research enabled by GENI will only exacerbate this problem.

For the same reason, learning-based *anomaly* detection systems—in which "normal" behavior is modeled using machine learning algorithms, and these models are then used to detect departures from "normal" behavior—are also unlikely to be broadly applicable in GENI.

Individual experiments will likely be too varied to enable the construction of a model of normalcy that is applicable to all of them (while still being useful for limiting unintended behaviors). Moreover, individual experiments may be too transient to establish a reliable baseline of "normal" behavior specific to that experiment, and even if such a baseline were established, it would likely include behavior that was not intended by the experimenters, owing to the immaturity of the experiment. As such, we anticipate that neither signature-based network intrusion detection nor learning-based network anomaly detection will be used extensively for monitoring network traffic in GENI.

In place of these methods, we anticipate requiring each PI to declare aspects of her experiment's network behavior in advance, so that this behavior can be examined and approved as reasonable, and so that the experiment can be monitored for compliance with that declaration. Intrusion detection based on specifications of intended behavior (versus specifications of bad behavior) is the domain of *specification-based intrusion detection*, e.g., [20][32]. Though not widely used in practice, we believe this approach is more suitable for GENI due to its customization of the specification to each individual experiment; this should provide for fewer, more accurate alarms. In addition, while not every alarm indicates an intrusion, it indicates a behavior that the experimenters apparently did not anticipate; as such, they learn something from the alarm, as well. Finally, specification-based intrusion detection complements the authorization framework discussed in Section 4 , in that a specification can both monitor the enforcement of access-control policy and limit behaviors that must be permitted but that should not be overused.

In the initial version of GENI, we anticipate that the specifications requested from experimenters will be coarse and focused on enabling GENI to contain the behavior of runaway experiments, e.g.: maximum bandwidth consumption; address ranges and ports to which connections might be opened from the experiment's slice; whether communication to and from the legacy Internet is required by the experiment; and the amount of storage that its output will consume. (Of course, GENI will then need to include monitors to detect violations of these specifications.) We envision that experience with GENI will drive support for richer and more fine-grained specifications in subsequent versions, if necessary.

While we believe that specification-based intrusion detection is well-suited for use in GENI, specifying behaviors of one's experiment is not an activity to which most researchers are accustomed, e.g., in the context of experiments on PlanetLab. There are several aspects of this mechanism that will impact its acceptance among the research community, including the ease with which the requested specifications can be divined from experiments and expressed, and the consequences of alarms. Alarms will presumably need to have consequences to the experiment or experimenter, but at the same time, experimenters will need to be dissuaded from providing specifications so weak as to be meaningless. As such, we would argue that these specifications need to be an input to the process by which experiments are approved to run on GENI.

Finally, while we expect network behavior to be monitored primarily via specification-based intrusion detectors, other types of behaviors may be more suitable for monitoring via more traditional approaches. For example, unexpected modification of files can be detected using the well-known Tripwire tool or a variety of existing rootkit detectors. More generally, we

recommend that audit trails be generated from a variety of vantage points and, when those vantage points are accessed via complex authorization relationships (e.g., chains of delegation, shared authority), that the proofs of compliance with access-control policy be recorded so that responsible parties can be discerned when problems are detected.

6.2 An Example Audit Trail Generator: PlanetFlow

Costs and requirements of an auditing system for GENI are difficult to predict, but some insight can be gleaned from the auditing system used in PlanetLab. This system, known as PlanetFlow, logs the packet headers of all packets sent and received by each node in the system. Packets are collected into flows, using the standard 5-tuple of <source IP, source port, destination IP, destination port, protocol>. These tuples are augmented with start and end times, total number of packets, total number of bytes, and the slice responsible for the traffic. In the case of UDP traffic, which lacks any easy determination of sessions, all UDP traffic from a single slice to the same destination port and address are considered part of the same session. It should be noted that this level of auditing does not keep track of the precise timing of each packet, other than the first and last packets within a session. In practice, this level of detail has not been needed in handling any complaint on PlanetLab. We should note that the GENI facility architecture group is studying ways to capture more detailed data, but for a different purpose—to serve the needs of those researchers studying how GENI is being used. Typically, this will require more intensive traffic monitoring, but for briefer periods. We refer the reader to the GENI Instrumentation and Measurement Systems Specification Document for more detail.

In terms of resource consumption, PlanetFlow's logging generates on the order of 2 Kbps (after compression) for every 1 Mbps of PlanetLab traffic. We believe this amount of overhead is acceptable, particularly since only traffic from GENI to the outside Internet needs to be tracked, and since complaints usually arrive promptly, there is no need to permanently store the logs. The processing overhead is similar: on average, PlanetFlow consumes 2.5% of each node's CPU on PlanetLab. We note this code is not heavily optimized. The main collector uses netlink sockets to capture packet information, which is then aggregated into flows by a user-level C program before being inserted into a MySQL database. Performance of this system has not been an issue for PlanetLab, and even if it were to scale linearly with bandwidth, it would not be a first-order barrier for GENI. There are some known sources of improvement possible:

- Have a kernel module aggregate packets into flows, which would reduce most of the data transfer and context switches to the user-space program. Especially for large data flows over high-bandwidth connections, this step would greatly eliminate the data volume between the components.
- Replace MySQL with a custom database specialized for the task at hand. Though the insertion process in MySQL has been known to be the major cycle consumer within PlanetFlow, it has not been worth replacing at this point because the overall resource profile of PlanetFlow has been low enough to be ignored. However, a more specialized database could reduce much of the overhead
- Switch from an eager architecture to a lazy one. Right now, all data is processed as soon as it is made available, even though the vast majority of it will never be queried. If a lazy approach were used instead, this data could be logged to disk immediately after the

aggregation step (or even without aggregation, if disk space is not an issue), with no other indexing or processing. Queries would take longer since the data would then have to be processed, but given that most data is never examined, the tradeoff can reduce PlanetFlow's profile tremendously.

Of course, the auditing system for GENI may require additional data beyond the current tuples. For example, it may be desired to have more than just a simple tuple for each session, with the options ranging from timestamping every packet sent/received to keeping more information from the IP/TCP headers, or even logging part/all of the packet. These extra options require more processing and storage, but can also provide a more complete view of activity from the node to other nodes. Conversely, it may also be desirable to filter information, such as all of the traffic between GENI nodes, or some subset of the traffic that matches other specified rules. Once these choices have been made, the processing and storage requirements can be calculated, and an appropriate prototype can be tested. While we expect that most of these choices should not cause too much extra CPU consumption, even at higher speed, any choice that requires higher storage may find itself competing for disk bandwidth or seeks with other applications. In these cases, the standard node configuration may opt to have a dedicated disk per cluster for storing the audit log.

7 Open Issues

Several open technical issues discussed in the previous sections must be addressed prior to an initial deployment of GENI. Moreover, looking at subsequent refinements of GENI beyond the initial deployment, we see opportunities to address other issues that we have not attempted to address yet (and that we do not expect to be addressed in its initial version). As this document is revised, such issues will be added to this section. Below is a partial list of open issues:

- In some instantiation of GENI (though probably not at initial deployment), it might be advisable to move the control plane of GENI away from IP infrastructure and toward one that offers better resilience to denial-of-service attacks. As discussed in Section **Error! Reference source not found.**, at present we accept the possibility of denial-of-service attacks against the GENI control plane and, in response, GENI will be designed to retreat to a safe state when its safe operation cannot be assured. Moving the GENI control plane to a more robust protocol suite should better minimize the circumstances in which such a retreat will occur, thus increasing the effective availability of GENI. That said, we avoid making a recommendation for a DoS-resilient control plane at this point in time, for multiple reasons. First, the design of DoS defenses is among the most actively pursued research topics in the network security community today, and it is characterized by a plethora of techniques and manners of defeating DoS. As such, we hope that one of the successes of the initial deployment of GENI is the testing and validation of various candidates for a more robust GENI control plane. (It is notable that such validation has been lacking to date, owing in large part to the lack of a facility like GENI.) Second, this control plane may itself utilize a protocol suite that is distinctly different from IP, and so the DoS defenses must work in concert with this protocol suite. This, too, is a topic to be worked out via competition on the GENI facility.

- The extent to which operational data in the GENI facility should be public information is yet to be determined. In the context of this document, one example of such operational data is access control policy and the authority that various parties possess. Retaining privacy of this information in the context of decentralized access-control systems such as that in Section 4 has been a topic of some study (e.g., [37][34][35]), though we believe that further study is needed to ascertain the circumstances in which such privacy policies would impose on the efficiency or even the possibility of completing access proofs in a system like GENI.

- More generally, the numerous operational practices and procedures needed to maintain the security of a facility like GENI (and to protect the Internet from GENI) also remain to be defined. These should obviously include industry best practices, e.g., keeping GENI nodes up-to-date with the latest patches; periodic "fire drills" to maintain a state of readiness for the operational staff to respond to large-scale events; and procedures for handling alarms and for propagating those alarms to others. Additional procedures need to be put in place, however, that are unique to GENI, not the least of which is some procedure for evaluating experiments (and accompanying requests for new permissions) as to whether they are acceptably safe. This is particularly true for experiments that involve malware, as security researchers may well want to conduct such experiments on GENI. The protection mechanisms described in this document, together with adequate intra-node isolation (e.g., via secure virtual machine monitors) not described here, could offer a basis for constraining such experiments to render them acceptably safe. But the means for making this decision and configuring the slice accordingly remains to be specified.

8 Conclusion

This document has described the requirements for security of the initial deployment of the GENI facility, and surveys the technologies that we believe should be developed to address those requirements that are specific to the GENI facility. These technologies include a pervasive access-control infrastructure for regulating access to GENI and experimental resources alike, and for enforcing an approximation of least-privilege access rights (Section 4); mechanisms for protecting the private keys that underlie this access-control infrastructure (Section 5); and the use of (primarily, specification-based) intrusion detection to monitor for experiments that perform outside the bounds that their experimenters expect and that have been approved (Section 6). Each of these technologies, if adopted for GENI, will need to be constructed for the GENI facility; we are unaware of commercial implementations of these components that would be adequate for adoption in GENI in their present forms. We believe they offer numerous benefits toward addressing the unique security requirements of the GENI facility. Combined with operational best practices, we believe that these technologies can render the GENI system to be both safe and usable by its target community of experimental network and distributed systems researchers, illustrating the benefits of architecting security from the beginning into a globally distributed system of networks and computers.

References

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security* 6(1–2):3–21, October 1998.

[2] T. Anderson (ed.). GENI Distributed Services. *GENI Design Document 06-24,* Distributed Services Working Group, September 2006.

[3] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, November 1999.

[4] J. Basney, R. Campbell, H. Khurana, and V. Welch. Towards operational security for GENI. GENI Design Document GDD-06-10, July 2006. Available at http://www.geni.net/GDD/GDD-06-10.pdf.

[5] L. Bauer, S. Garriss and M. K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 81–95, May 2005.

[6] L. Bauer and M. A. Schneider and E. W. Felten. A general and flexible access-control system for the Web. In *Proceedings of the 11th USENIX Security Symposium*, August 2000.

[7] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 253–266, March 2004.

[8] M. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 139–154, 2004.

[9] M. Blaze, J. Feigenbaum, J. Ioannidis and A. D. Keromytis. The KeyNote trust management system, version 2. IETF RFC 2704, September 1999.

[10] M. Blaze, J. Feigenbaum and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[11] E. Belani, A. Vadhat, T. Aderson and M. Dahlin. The CRISIS wide area security architecture. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[12] D. Boneh, X. Ding, G. Tsudik, and C. M. Wong. A method for fast revocation of public key certificates and security capabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 297–308, August 2001.

[13] C. Boyd. Digital multisignatures. In *Cryptography and Coding*, pages 241–246, Clarendon Press, Oxford, 1989.

[14] C. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas and T. Ylonen. SPKI certificate theory. IETF RFC 2693, September 1999.

[15] Y. Fu, J. Chase, B. Chun, S. Schwab and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[16] R. Ganesan. Yaksha: Augmenting Kerberos with public key cryptography. In *Proceedings of the 1995 ISOC Network and Distributed System Security Symposium*, pages 132–143, February 1995.

[17] J. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. *Journal of Computer Security* 9:47–74, 2001.

[18] ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework, June 1997.

[19] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, May 2001.

[20] C. Ko, M. Ruschitzka and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, May 1997.

[21] B. Lampson. Computer security in the real world. In *Proceedings of the Annual Computer Security Applications Conference*, 2000.

[22] B. Lampson, M. Abadi, M. Burrows and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems* 10(4):265–310, November 1992.

[23] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. *International Journal of Information Security*, 2004.

[24] P. MacKenzie and M. K. Reiter. Networked cryptographic devices resilient to capture. *International Journal on Information Security* 2(1):1–20, November 2003.

[25] P. MacKenzie and M. K. Reiter. Delegation of cryptographic servers for capture-resilient devices. *Distributed Computing* 16(4):307–327, December 2003.

[26] A. Nicolosi, M. Krohn, Y. Dodis and D. Mazieres. Proactive two-party signatures for user authentication. In *Proceedings of the 10th ISOC Network and Distributed System Security Symposium*, pages 233–248, February 2003.

[27] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed resource discovery on PlanetLab with SWORD. In *Proceedings of the First Workshop on Real, Large Distributed Systems*, December 2004.

[28] S. Pearson. *Trusted Computing Platforms: TCPA Technology in Context*. HP Professional Series, Prentice Hall, first edition, 2002.

[29] L. Peterson, A. Bavier, M. Fiuczynksi and S. Muir. Experiences building PlanetLab. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation*, November 2006.

[30] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[31] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE* 63(9):1278–1308, September 1975.

[32] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang and S. Zhou. Specification-based anomaly detection: A new approach for detecting network intrusions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 265–274, November 2002.

[33] W. Trappe. Requirements Document for Security of GENI Wireless Networks. *GENI Design Document 06-16*, Wireless Working Group, September 2006.

[34] W. H. Winsborough and N. Li. Safety in automated trust negotiation. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[35] M. Winslett, C. C. Zhang and P. A. Bonatti. PeerAccess: A logic for distributed authorization. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[36] E. Wobber, M. Abadi, M. Burrows and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems* 12(1):3–32, February 1994.

[37] T. Yu, M. Winslett and K. E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. ACM Transactions on Information and System Security 6(1):1–42, February 2003.