Spyglass: Demand-Provisioned Linux Containers for Private Network Access

Patrick T. Cable II, Nabil Schear
MIT Lincoln Laboratory
{cable,nabil}@ll.mit.edu

Abstract

System administrators are required to access the privileged, or "super-user," interfaces of computing, networking, and storage resources they support. This low-level infrastructure underpins most of the security tools and features common today and is assumed to be secure. A malicious system administrator or malware on the system administrator's client system can silently subvert this computing infrastructure. In the case of cloud system administrators, unauthorized privileged access has the potential to cause grave damage to the cloud provider and their customers. In this paper, we describe Spyglass, a tool for managing, securing, and auditing administrator access to private or sensitive infrastructure networks by creating on-demand bastion hosts inside of Linux containers. These on-demand bastion containers differ from regular bastion hosts in that they are nonpersistent and last only for the duration of the administrator's access. Spyglass also captures command input and screen output of all administrator activities from outside the container, allowing monitoring of sensitive infrastructure and understanding of the actions of an adversary in the event of a compromise. Through our evaluation of Spyglass for remote network access, we show that it is more difficult to penetrate than existing solutions, does not introduce delays or major workflow changes, and increases the amount of tamper-resistant auditing information that is captured about a system administrator's access.

1 Introduction

System administrators have super-user access to the low-level infrastructure of the systems and networks they maintain. To effectively do their job, they need to access the sensitive interfaces of switches, routers, operating systems, firmware, virtualization platforms, security appliances, etc. We rely increasingly on this infrastructure for tasks, from ordering food to controlling complex mechanical systems like the electric grid. Given the typical administrator's breadth of access to this infrastructure, administrators or the client devices they use are a prime target for compromise by a motivated adversary. Alternatively, if the administrator and the adversary are the same (i.e., a rogue administrator or insider), then this administrator often has unchecked access to disable and evade the security controls of the network.

To protect the sensitive interfaces an administrator must use, the system architect can place these interfaces on a private network or VLAN that is not broadly accessible to either the Internet or even and organizational LAN. This practice is also commonplace in Infrastructure-as-a-Service cloud environments at both the tenant layer (e.g., the user of virtual machines) and the provider layer (e.g., the operator of the virtual machine hosting environment) [27]. Firewalls, virtual private networks (VPNs), and bastion hosts allow remote access for the administrator into the sensitive network. Firewalls and VPNs open new security vulnerabilities by directly connecting a potentially untrusted client system directly to the sensitive network, and the do not directly offer an audit log of the administrator's activities. Bastion hosts explicitly isolate the client system from the network and offer a centralized place to audit activities. However, bastion hosts themselves can be compromised, leading to a catastrophic security collapse where the adversary can impersonate *any* administrator and wreak havoc across the network.

To address the security shortcomings of bastion hosts, while retaining good network isolation and audit capabilities, we created Spyglass. Spyglass is a tool that provides on-demand nonpersistent bastion hosts to each administrator to facilitate access to sensitive networks. The

This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

system creates a Linux container using Docker for each user's session and destroys it after the user disconnects. Through a least-privilege system design, Spyglass lowers the risk of compromise to the bastion server itself. While Spyglass does not prevent insiders with valid credentials from accessing the sensitive network, it does provide a tamper-resistant audit record of their activities. This capability allows an organization to forensically track the moves of adversaries and assists in recovery and cleanup.

This paper's primary contributions are:

- Design for securely isolating and monitoring the actions of system administrators while reducing the threat posed by insiders and phishing attacks
- Implementation of the Spyglass prototype and best practices for deployment
- Security and performance evaluation showing that Spyglass is more difficult to penetrate than previous solutions and that it can be implemented without considerable delay or workflow changes.

The rest of this paper is structured as follows: Section 2 describes the the problem, threat model, and existing solutions. Section 3 discusses the design of the system. In Section 4, we describe the components of the system and their implementations. We evaluate both the performance and security in Section 5. Section 6 reviews related work. We discuss the current status of Spyglass and opportunities for future work in Section 7, and conclude in Section 8.

2 Background

A system administrator often connects to a variety of interfaces to perform their work. These interfaces may be used to configure switching or routing logic, or to access hardware in the event of a system crash. The administrator may connect to the host running a virtualization platform, or a machine instance operating a cloud platform. Given the success and prevalence of DevOps environments, the administrator may also be making code changes to the software that actually runs the provider's self-service platform.

It is easy to see why administrator credentials are so sought by adversaries, either those looking to compromise an administrator for an organization, or a software-as-a-service customer of that organization. Credential theft can be crippling: in June 2014, an adversary compromised the Amazon Web Services (AWS) credentials of CodeSpaces, a company that provided cloud-based source code repository hosting. The adversary then asked the company for a sum of money by a certain time. When the money was not paid, the adversary then deleted all of CodeSpaces' AWS instances and disk storage, along with all of their backups. The company folded shortly thereafter [10].

One of the most popular ways to obtain credentials is by phishing. In the most damaging phishing attacks, an adversary convinces an administrator to install malware on their computer, steals their credentials, and then spreads across the network that the administrator maintains. Some of the most serious breaches of 2014, including those on Sony Pictures [3] and JP Morgan Chase Bank [2], involved the theft and misuse of administrative credentials. Indeed, these attacks were most damaging precisely because of this fact.

Given that our infrastructure can be compromised by either an inside or outside adversary, we need a solution to limit the impact of these attacks. As part of security best practices, the networks on which the most sensitive of these interfaces are hosted are often separated from public-facing or even internal LANs. This limits the accessibility of these sensitive interfaces and protects the credentials for accessing them from eavesdropping.

Since administrators must invariably access these isolated networks to do their work, we need ways to facilitate remote access. The goals of an ideal remote access solution should provide security for remote access, strong authentication, and audit logging of all actions that take place across the trust boundary. In the following sections, we describe the existing remote access solutions and their strengths and weaknesses with respect to this set of goals.

2.1 Firewalls

When a sensitive network is firewalled off from an untrusted network, the firewall allows or denies traffic based on policy rules. This provides a layer of security to the protected network. Hosts exposed to external networks need to contend with malicious traffic, many of which attempt to brute-force common passwords or attempt known attacks en masse to any host that will listen. The firewall allows for a central focus point for security decisions, and enforcement of security policy [37]. Indeed, "firewalls are an important tool that can minimize the danger, while providing most – but not necessarily all – of the benefits of a network connection" [1].

The downside of firewalling traffic is that it only allows network-based filtering of traffic. *Firewalls* do not establish authorization of a user to connect. They do not protect against IP spoofing attacks. Multiple users could be behind an IP address that is chosen as an appropriate host from which to receive traffic. This leaves the authorization decision to the remote device. Firewalls also do not protect against a trusted insider. Similarly, they do not do anything for the remote host in the sensitive

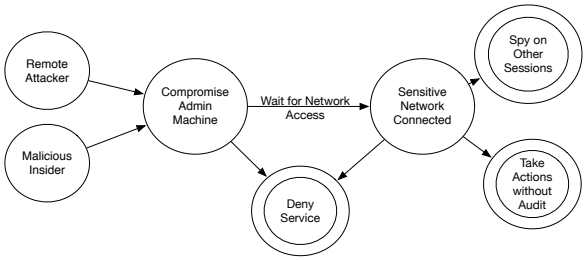

Figure 1: A state diagram showing steps required to compromise a sensitive network protected with Firewalls and VPNs.

network:

> Given that the target of the attackers is the hosts on the network, should they not be suitably configured and armored to resist attack? The answer is that they should be, but probably cannot. Such attempts are probably futile. There will be bugs, either in the network programs or in the administration of the system. [1]

Indeed, there are bugs. For example, many administrators use the Intelligent Platform Management Interface (IPMI) to perform remote administration functions. This protocol and the hardware that implements it are both critical to the ability to remotely debug system failures and problems. Yet, one independent researcher found that close to 90% of implementations that were publicly accessible had a security issue that would allow unauthorized access to the hardware [7]. Some of these expose their password by querying a device using Telnet [34].

We show a state diagram in Figure 1 that illustrates what an attacker would have to do to compromise the sensitive network. A firewalled network may always be connected to the host. This reduces the amount of time an adversary may have to wait to compromise the sensitive network.

2.2 Virtual Private Networks

A popular methodology for separating sensitive and untrusted networks is to place a host between two networks. In the VPN methodology, the host runs software such as OpenVPN that facilitates a remote host "joining" the network as if it were there locally [25]. Many organizations use this to facilitate remote workers: the worker can be anywhere, and the traffic between the company and the end user's laptop is encrypted to prevent the data from interception or eavesdropping.

The ubiquity of the VPN is due in part to its ease of use. A user installs a client application configured by their organization, and is able to connect to the network and access the network in its entirety. Applications don't need to be redesigned to deal with external access, and an organization can rest assured that most of their data stays on the internal network.

However, in the era of the "French-bread model" of network security, this has meant that an external laptop has unfettered access to the soft inside of the network [13]. This makes the administrator's laptop a perfect pivot point to infiltrate a network that connects to sensitive infrastructure. Many organizations attempt to deal with this risk via policy. For example, policies like "Establish a VPN connection immediately after establishing Internet connectivity" and "do not connect any non-work-owned devices to a work-owned laptop" are common. There are multiple reasons, intentional and unintentional, that may cause an employee to not follow the rules. For example, an employee may connect to a malicious wireless access point. The owner of the malicious access point may inject advertisements that are provided by a malware carrier, infecting the computer.

Finally, while logging and auditing of VPN connections themselves is straightforward to implement at the VPN concentrator, correlation of a user activities through network logs, host logs, and authentication information is more challenging. First, the VPN connection will virtually connect the remote user to a dynamically chosen IP address within the sensitive network that may have previously been used by another VPN user. Second, the VPN user's activities on the host (e.g., commands executed or data copied) must be combined with network logs to understand the impact of a malicious actor.

2.3 Bastion Hosts

Bastion hosts are like the lobby of a building: "Outsiders may not be able to go up the stairs and may not be able to get into the elevators, but they can walk freely into the lobby and ask for what they want" [37]. Bastion hosts provide a single point to *audit* traffic as an interface that can be controlled by the organization that owns or controls the private network, as opposed to just being able to see basic network flow data (source, destination, session duration, etc.). Firewalls and VPNs allow you direct access to a remote network, without having to necessarily "check in."

Providing a controlled interactive session, as opposed to firewalled or VPN-based access, carries benefits for the organization that controls the sensitive network. The organization does not have to worry *as much* about the state of the administrator's workstation. The organization can employ software and methods used to monitor workstations and integrate these with existing security infrastructure. Figure 2 shows that compromise of the sensitive network is more difficult with a bastion host than with firewalls or VPNs.

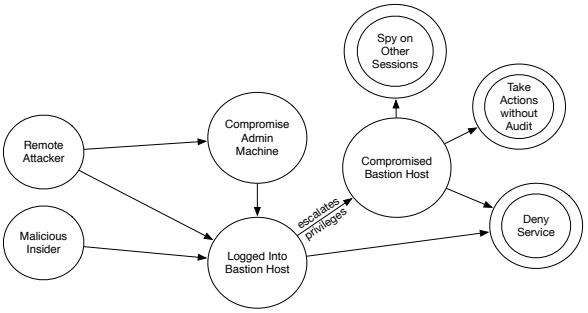

Figure 2: A state diagram showing steps required to compromise a sensitive network protected with a traditional bastion host.

Bastion hosts provide a centralized point at which to enforce strong authentication and to capture detailed audit logs of user activity. Their primary weakness is in the new vulnerabilities they introduce. The act of providing an interactive session on the bastion host to an end user is risky. In the case of a malicious insider, the organization has given logical access to a bastion host; if any pieces of software on the bastion host are compromised, the insider can attribute actions to other users, get a set of password hashes of other accounts, and/or key log to gain access to other devices on the sensitive network. Literature going back decades covers how attackers break into bastion hosts and create persistent environments [4].

3 Design

We believe the bastion host pattern provides the best solution to achieve secure and audible remote access for system administrators. To implement a secure bastion host, we need to address the weaknesses in typical bastion deployments like single point of failure, tamperable audit information, and weak passwords.

Our goals in this work are to minimize the risk of the bastion itself, while providing higher security for system administrators and the isolated networks they use. We want to have the ability to audit and log, in a tamper-resistant manner, all activities that a user makes on the sensitive network. We want to limit the spread of an external attacker and the impact they can have. Finally, we want to ensure that even if they do compromise the sensitive network, we can recover using the audit log.

To address these challenges and our set of goals, we developed Spyglass. Spyglass is a network access device that is dual homed on an untrusted network and sensitive network where the interfaces to critical security infrastructure reside (see Figure 3). A user wishing to access the sensitive network authenticates to Spyglass and requests a nonpersistent, isolated session. From this session, the user can access resources on the sensitive network. From a vantage point outside of the user's session, Spyglass monitors and records all of the user's activities. In the following sections we review the threat model for Spyglass and then discuss its four key design components: multifactor authentication, isolation, nonpersistence, and auditing.

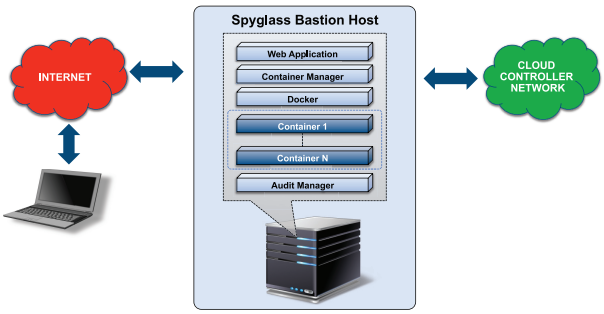

Figure 3: Spyglass System Design

3.1 Threat Model

We assume that the adversary is either a malicious system administrator or that the system administrator's client system has been compromised. The goal of the adversary is to compromise an isolated network the system administrator controls. We assume that the adversary may be able to compromise applications inside of the containers that face the untrusted network. However, we assume the adversary cannot break out of the sandbox Spyglass creates, and cannot compromise the control process for creating and destroying containers. We assume that valid users have used multiple factors to authenticate to the system creating the containers. Finally, we assume that SSH is properly configured (i.e., disabling tunneling) along with the network bridge device used by the OS-level virtualization provider.

3.2 Multifactor Authentication

We begin by considering how to authenticate Spyglass users. Reusable passwords are both easily cracked and easily stolen [21]. Indeed, with custom hardware, an adversary can crack passwords at a rate of 350 billion guesses per second [9]. Adding multiple factors makes it more difficult to steal a user's credentials to obtain unauthorized access. Some types of multifactor authentication require that the valid user be physically present to initiate a session. In the case of an administrator whose client system is compromised by an external attacker, this slows the attacker to only be able to initiate a session when they can subvert one initiated by the valid user.

Best practices for organizational cyber security also agree on the importance of multifactor authentication. For example, the SANS Institute's Critical Security Con-

trols recommend that multifactor authentication should be used "for all administrative access, including domain administrative access" [30]. Unfortunately, some assets (e.g., networking or storage appliances) cannot take advantage of multifactor authentication natively. By introducing the Spyglass bastion in front of these systems, we are able to better address the SANS control's recommendation.

Many government institutions implement multifactor authentication by taking advantage of the cryptographic functions of smart cards. For organizations that already have the infrastructure required to operate a large smart card infrastructure, this could be suitable for Spyglass. However, the implementation often requires additional hardware (both in the form of the cards and readers). It also requires complicated integration to allow for those capabilities to be used to authenticate with websites.

To address these scalability and adoption challenges, we chose Yubico YubiKey to add another authentication factor for Spyglass authentication [36]. The YubiKey is a USB device that outputs a 44-character string of ASCII letters that represent a 12-character identifier and a 32-character one-time password based on the secret and public identifier stored on the hardware device. A large community exists around the use of the YubiKey, and an open-source YubiKey Validation Server exists along with cross-language libraries to interface with the server.

3.3 Isolation

While it's certainly easier for a user to directly connect to a sensitive network (either via firewall or via VPN), as discussed in Sections 2.1 and 2.2, it comes at a cost to the security posture of the sensitive network. Malware on the users system may have unfettered access to the sensitive network and may directly connect to and attack hosts there. For this reason, Spyglass, as other bastion hosts do, explicitly isolates and separates the administrator's computer and the sensitive network. Isolation is critical, since "[t]oday's cyber incidents result directly from connecting formerly standalone or private systems and applications to the Internet and partner networks" [8].

We also introduce isolation between the different users of Spyglass and the components that underpin Spyglass. Thus, each user gets their own login environment from which to pivot to the sensitive network and, similarly, each component of Spyglass is in an isolated environment and only communicates to other components over minimal well-defined interfaces. Traditionally, virtualization provides an answer for system architectures that required that two subsystems couldn't necessarily affect each other's memory space in unexpected ways. However, that assurance comes at a performance cost. Creating a virtual machine for each user would pose a significant resource overhead and delay considering that virtual machine spin-up times (even in the cloud) exceed 30 seconds regularly.

To achieve strong isolation without the performance overhead of full system virtualization, we utilize OS-level virtualization technologies to isolate Spyglass components and users. This method of virtualization allows multiple environments to share a common host kernel and utilize underlying OS interfaces, thus incurring less CPU, memory, and networking overload [29].

3.4 Nonpersistence

Increasing the amount of ephemerality in the system design works to the organization's advantage in defending their systems. Goldman found the benefits that nonpersistence provides makes an attacker's job more difficult. Specifically, consider the ability it allows organizations to stand up and tear down a particular capability (in our case, remote access) in an on-demand fashion, and the ability to ensure that a particular state is regularly patched [8].

To understand the importance of nonpersistence, we need to understand the kill chain of an attack. The Cyber Kill Chain describes the steps an attacker must take to compromise a computer system. Generally, to launch a successful attack on a system, an attacker must collect useful information about a target, attempt to access the target, exploit a vulnerability for the target, launch the attack, and then find a way to maintain access to the system [24]. Nonpersistence interrupts an attacker's ability to persist by forcing session timeouts and subsequent destruction of their environment.

In Spyglass, new user sessions are always instantiated inside of a fresh container that is patched regularly. Even if an attacker can compromise the container, they will have to repeat this process regularly and potentially raise their profile in other monitoring and logging capabilities of the system, leading to a higher chance that an attacker will be detected.

3.5 Audit

The presence of some form of situational awareness when it comes to running a server that is available on an untrusted network like the Internet is an important asset. It is otherwise impossible to know whether a compromise has occurred if there isn't a means to audit and monitor accesses, user actions, and other items of interest. Considering that it is not a matter of*if* one gets hacked but rather *when* [37], it makes sense that seeing an attacker's actions that allowed them to compromise the host would aid in repair and recovery.

While often given less importance than active security measures like strong passwords or antivirus, best practices include the need for audit logging. The Australian Signals Directorate recommends "centralised and time-synchronised logging of successful and failed computer events" and "centralised and time-synchronised logging of allowed and blocked network activity" as part of their *Strategies to Mitigate Targeted Cyber Intrusions* report. Specifically:

> Centralised and time-synchronised logging and timely log analysis will increase an organisation's ability to rapidly identify patterns of suspicious behaviour and correlate logged events across multiple workstations and servers, as well as enabling easier and more effective investigation and auditing if a cyber intrusion occurs. [5]

Similarly, MITRE's report *Building Secure, Resilient Architectures for Cyber Mission Assurance* specifies detection and monitoring as one of five objectives that help achieve architecture resilience:

> While we cannot always detect advanced exploitations, we can improve our capabilities and continue to extend them on the basis of after-the-fact forensic analysis. Recognizing degradations, faults, intrusions, etc., or observing changes or compromises can become a trigger to invoke contingency procedures and strategies. [8]

These ideas lead to the requirement that a system be in place that captures all commands issued and their output for later retrieval and review. These logs need to be located on a remote host to preserve their content in the event that the bastion host machine is compromised. In Spyglass, we further protect the logs from tampering by capturing and transmitting the audit log information from outside of the user's container. Furthermore, if the attacker is able to disrupt the logging process somehow, the session is immediately terminated. This leads to a system where a system administrator is unable to take any actions on the sensitive network without leaving a trail of what they did.

4 Implementation

Spyglass consists of four components: a locally hosted YubiKey validation server, the Spyglass web interface, the container daemon, and the audit daemon. To maintain proper segmentation, the YubiKey Validation Server and the database server should be on a separate VLAN that is not on the sensitive network pictured above. Another independent host on a separate VLAN should store audit log data. Additionally, this audit host should be controlled/maintained by parties other than the system administrators using Spyglass (e.g., by a security policy or oversight rather than IT organization) to avoid the possibility of audit log tampering.

4.1 User Facing Interface

The user interface for Spyglass is required to:

- Authenticate a user
- Store a valid SSH public key for each user
- Instantiate a bastion container
- Destroy a bastion container
- Be accessible from a variety of client platforms

Figure 4 displays the Spyglass architecture, along with numbers representing relevant communication flows. During session initiation, a user (1) accesses the web application. The application (2) checks the authentication against a database and validates the other factor. Once the user is logged in, they (3) request a container and the web application sends a request to the container daemon with information about the user and the preferred key. The container daemon pulls this information from the database in step (4), and sends this information to Docker in step (5). Docker then (6) creates the container and sends information back to the web application to inform the user what host their container is running on. Finally, the user logs into their container in step (7). This creates a set of log files, which are read by the audit daemon and moved to the audit host (8). Processes that run on the bastion host are outlined with a dotted line.

Upon initial login, Spyglass presents users with the main interface in Figure 5. The user then adds an SSH public key by going to the *Keys* menu and clicking *New Key*, as seen in Figure 6. Once the key is added, the user can now start a session by going to the *Sessions* menu and clicking *New Session* as seen in Figure 7. Afterwards, Spyglass presents the user with session information (Figure 8). The user can now initiate an SSH connection to the bastion container and access the sensitive network.

4.2 Container Daemon

We need to enable the web UI to handle container management through Docker. Rather than doing this directly from the web application, we chose to implement a middleware process called the container daemon (or `containerd`). The primary motivation for this design was to avoid giving the Spyglass web application root privileges so that it could access the Docker control

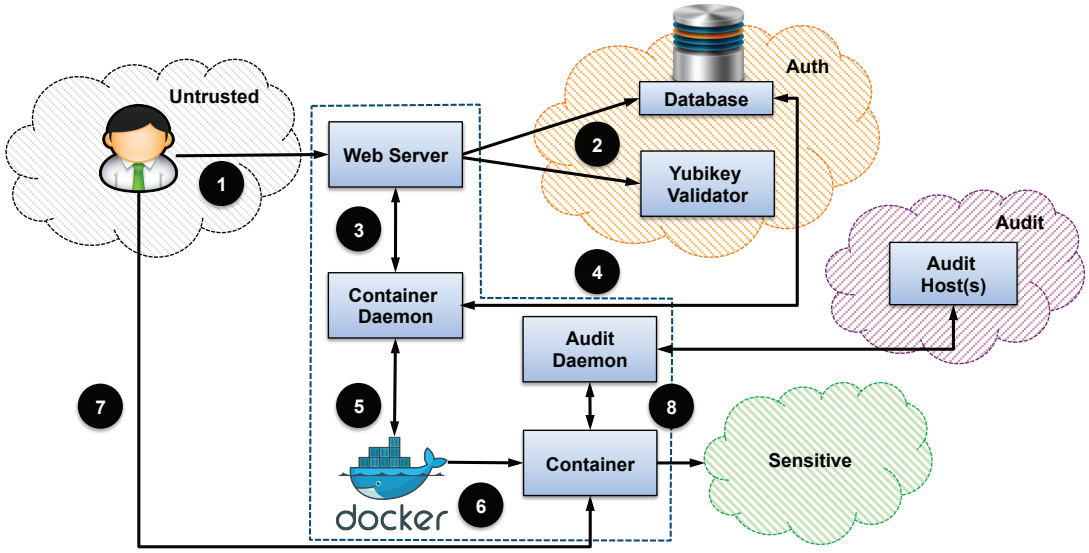

Figure 4: Spyglass infrastructure and information flows.

socket (owned by `root`). The separation also enables us, in future work, to further isolate the web application from containerd by placing the web application itself inside of a container and using SELinux mandatory access controls to implement least privilege.

We wrote containerd in a strongly typed language (Go) to prevent a variety of simple attacks on the web application itself. Containerd supports a small, simple, and well-defined interface for commands that further limits its attack surface. Thus, even if the web application is compromised, it will only able to create and delete containers and not access any other user's session.

The container daemon provides two HTTP Endpoints: `containercreate` and `containerdestroy`. The `conatinercreate` endpoint handles the creation of a container via an HTTP POST. It expects to receive a JSON object that references a database that the container request application uses.

Listing 1: A sample JSON container created notification

```
{
  "DbKeyId":10,
  "DbUserId":2,
  "SshKey":"ssh-rsa AAAAB3NzaC1yc2EAA<truncated>",
  "SshUser":"cable",
  "SshPort":"49154",
  "DockerId":"c46a32bd3347<truncated>"
}
```

Upon receiving the request, `containerd` queries the request application's database to get the appropriate username and SSH public key needed to insert into the container. Once it has the appropriate metadata, containerd instantiates a `docker` object with the appropriate configuration for the container. After Docker creates and starts the container, containerd returns a JSON object with the information about the container to the request application (as Listing 1 shows).

The `containerdelete` endpoint handles the deletion of a container via an HTTP DELETE. It only accepts a container identifier, and passes it to Docker for deletion.

4.3 Audit Daemon

Initially, we evaluated SSLsnoop for use in the system to capture activity inside the bastion containers from a vantage point outside the container boundary [20]. SSLsnoop locates the SSH session keys in the SSH process memory and does real-time decryption of traffic between two hosts. However, later versions of SSH have changed the format of in-memory structures, causing SSLsnoop to be unable to proper locate the key and encrypted stream. SSLsnoop also only monitored the SSH connections originating from the bastion itself, so an attacker intent on breaking the container would go undetected.

To keep as much of the monitoring infrastructure outside of the container as possible, we settled on a hybrid solution using SudoSH and a custom log monitor. SudoSH works by spawning the user's shell inside of an environment that is transparently capturing keystrokes and screen output [12]. We are able to look into the container host's file system and use Linux's `inotify` functionality to read the logs from the container host and relay them to the audit host [23]. The `Audit Daemon` ensures that the logs are sent to the audit host regularly and thus avoid any tampering from compromise of a Spyglass container. To further ensure the integrity of the auditing system, Spy-

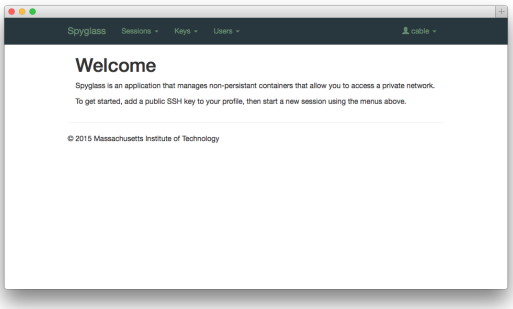

Figure 5: Initial login screen.

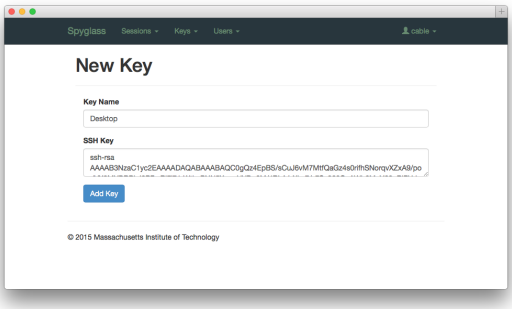

Figure 6: Adding an SSH key.

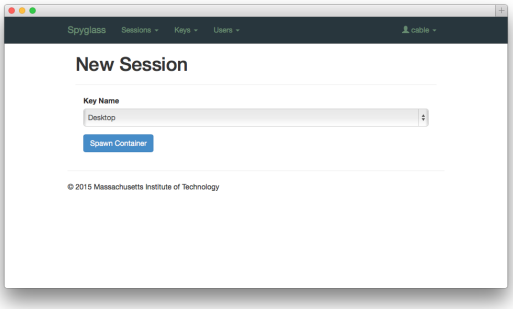

Figure 7: Creating a session.

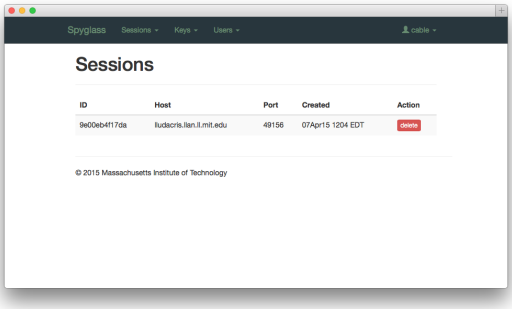

Figure 8: Viewing active sessions.

glass monitors the process table and will immediately terminate the user's bastion container if the SudoSH process stops.

5 Evaluation

We hypothesize that fast, on-demand provisioning of Linux containers that are unique to a particular user's session loosens the coupling between the integrity of the private network and the integrity of the remote client that connects to it. This separation is easy to provide as a service in part due to the lightweight nature of containers.

To prove this point, we analyzed the individual overhead of five containers on the host machine. We also attempt attacks on the system and attempt to connect to the authorization and auditing networks, along with some attempts to evade of the audit logging process. In Figure 9 we also created a state diagram, similar to Figures 1 and 2 to illustrate how Spyglass differs from existing firewall, VPN, and bastion host solutions.

These experiments were performed in VMware Fusion 7 Pro running on a Macbook Pro with 16 GB of RAM and a 2.6 GHz Intel Core i7 processor. The bastion host virtual machine has one processor core, and 1024 MB of memory.

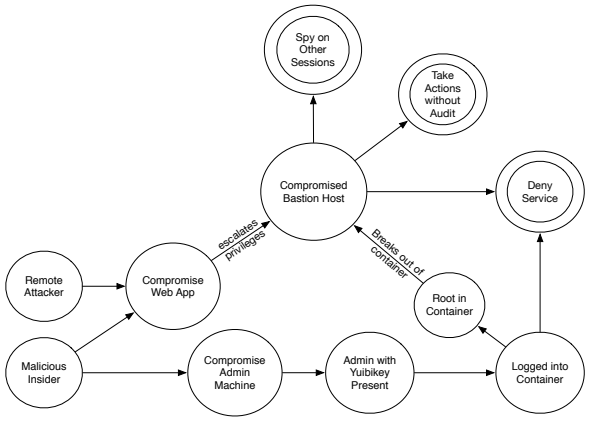

Figure 9: A state diagram showing steps required to compromise a sensitive network protected with Spyglass

5.1 Bastion Container Performance

To measure the load characteristics of an individual container, we used the Google tool cAdvisor [11] running on the bastion host. cAdvisor captures CPU and Memory load and writes it to InfluxDB, a time series database [17]. We monitored five invocations of the container that the container request application would instantiate. This session connected to a remote host and

ran the `top` command. We used the `time` command to measure the instantiation time of an individual container. We queried InfluxDB for `max(memory_usage)` and `last(cpu_cumulative_usage)` values for each individual container. Table 1 shows the results.

As expected, we find overhead and instantiation latency to be substantially lower than a virtual machine-based approach, where memory, instantiation, and CPU overhead are much larger [15]. We expect that even an embedded system with a low amount of RAM could support 10s of users. As a control channel for a sensitive network, we do not anticipate that Spyglass would limit network bandwidth.

5.2 Host Denial of Service

A serious potential attack that an adversary could launch is to deny service to other clients connecting to the Spyglass. To test this, we spawned a new container and ran a command to fill the container disk (`dd if=/dev/zero of= temp`). This in turn caused the host disk to fill. The system was still able to spawn new containers after the disk was full; however, their auditing processes were quickly killed off as the host ran out of disk space.

One solution is to use the `devicemapper` backend for Docker container storage. Using the `devicemapper` backend allows for finer-grained control on storage by specifying a base size for all container images. However, this means that all containers must be the same size; by default this value is 10 gigabytes. This effect can be mitigated by starting the Docker daemon with the `--storage-opt dm.basesize=1G` option; however, this breaks compatibility with the container creation web interface. Work towards user namespaces and individual quotas will make it trivial to apply file system quotas to containers; however, these features are not yet available.

5.3 Network Protection

The container host is connected to two different networks that are used to provide authentication and audit log storage support for the system. These networks should not be exposed to the user who is looking to access the protected network. By default, the container does not have access to use the `ping` command. However, it was still possible to use `netcat` to send data between two hosts if the destination address was known. It was also possible to connect back out to the untrusted network, which would allow an attacker to pivot to another host.

We implemented firewall rules on the Spyglass container host to mitigate these and other network-based attacks. This way, traffic to and from the container was limited to SSH inbound and a select set of outbound protocols for the container. Finally, the host firewall explicitly drops and logs all connection attempts from the container to the host. In our testing, we found no unauthorized network connections were allowed.

5.4 Container Escalation and Escape

A core assumption of the security of our system is that a user cannot escalate privileges and/or "break out" of the container itself. We accomplish this by proper configuration and multilayered defensive practices.

We configure the container in such a way that a remote user does not have root privileges. This is to protect against an escape attack within system, as it is easier to jump from the container to the container host if an attacker has root inside the container. We also suggest regular rebuilding of the container with patched binaries. This makes it more difficult for an attacker to take advantage of a root exploit in any base packages.

The use of mandatory access control can also limit the scope of an attacker who is able to both escalate to a root user within the container *and* break out of the container itself. Docker has SELinux rules available for use with Red Hat Enterprise Linux 7 and derivatives, but our implementation uses Ubuntu. In future work, we plan to implement this addition protection that would further raise the bar for an adversary trying to compromise Spyglass.

5.5 Audit Security

The logs created by the SudoSH process running inside of the container are ephemeral. To address this issue, we send the logs to another host on a separate network to provide a record in the event of a container compromise or other security event. `rsync` provides functionality to move files over to the audit host. We discuss methods to optimize this approach in Section 7.

6 Related Work

There is a variety of work that show early interest and effort into implementing container-based solutions to insulate a host operating system from attack. Ioannidis et al. implement a tag that is attached to files obtained from remote sources that allows built-in limiting what malicious code can do to a user's other files [19]. This is interesting, in that modern operating systems have implemented a variation of this idea (Apple's Mac OS X is able to detect files that have been downloaded and warn before opening); however, the technology that is more applicable to this project has gone largely unimplemented in major operating systems. Wagner also shows

#	Memory Use	CPU Cycles	Real Time	User Time	System Time
1	4.80	320437210	0.77	0.01	0.02
2	4.80	246014871	0.85	0.02	0.03
3	4.91	464523389	0.16	0.00	0.00
4	4.79	417975143	0.16	0.01	0.00
5	4.80	332404388	1.05	0.01	0.00

Table 1: Memory (*MB*), CPU (*jiffies*) and Time (*seconds*) for container instantiation.

early interest in the idea of containerization, and implements a method of attempting to "containerize" an application in user space by monitoring system calls [33]. Wagner monitors system calls and the files they act upon against a policy to ensure that applications are allowed to access specific files or network devices. The approach comes about a year before the release of SELinux, which uses contexts rather than per-application configuration to enforce access to resources.

Thakwani proposes a new UNIX `dfork()` call that instantiates the child process in a virtualized machine [32]. This solution is elegant in that it provides a very low-level means to ensure that processes start in separate namespaces. Thakwani's work doesn't measure the amount of time it takes to use `dfork()` with a new virtual machine on each use. Many processes can be sandboxed in the same virtual machine in Thakwani's architecture, thus saving time; however, this would not work well for our goal of isolating users from each other.

Parno et al. demonstrates demand-based virtualized containers that are instantiated upon user-login to a website in CLAMP [26]. CLAMP goes on to actively broker access to a particular database and ensure each container instance only contains the appropriate data for the authenticated user. While our work does not deal with specific user data, CLAMP demonstrates a model of mitigating risk by implementing nonpersistence and containerization.

Similarly, Huang et al. propose a framework to reduce an adversary's ability to have an attack persist on a particular network by refreshing to a known clean state on a regular basis [14]. This methodology works well on detectable and undetectable attacks thanks to the regular refresh interval. However, it does not protect against any lower-level (i.e., hardware) attacks that may occur [24]. It also provides some form of "highly available" architecture to handle the hosts that are being actively refreshed. Spyglass makes no guarantee of a highly available resource, but new containers are easy to instantiate unless the container host has failed.

Our approach is similar to the Lightweight Portable Security [22]. Lightweight Portable Security creates a bootable, read-only environment that doesn't store state. This affords an organization reasonable assurance that there is no persistent malware on a machine they may not own, which addresses concerns in Section 2.2 regarding virtual private networks. However, the technique has a significant amount of overhead in that it requires a user to reboot into the environment, and it makes no assumptions about attacks that would live in hardware (and therefore, persist across reboots) [24].

Nonpersistence can have operational benefits as well. An example of this is Ganger, a tool for instantiating containers when a network request is received [31]. The motivation for Ganger was to create a temporary environment that would ensure that files created under `/tmp` would be cleaned up in an orderly fashion after the network connection was closed. This was due to the use of a particular application that wrote a large amount of temporary data.

Proving that there is a market for monitoring of the connection concentrator, Pythian's *Adminiscope* implements a form of connection concentrator to a private network with live auditing ability [28]. However, it is unclear as to what mechanisms are implemented to guard the host against compromise and other threats to the concentrator itself. Similarly, another industry product exists named Invincea [18]. Invincea brings together concepts of non-persistence and isolation to protect a browser against web-based malware. In our system, we aim to protect sensitive infrastructure from a bad client.

A similar commercial offering is Dome9's Secure Access Leasing product [6]. Secure Access Leasing is a mechanism by which users request access to various cloud-hosted resources, and the Dome9 product has an agent that configures hosts and AWS firewalls to allow a particular user access to the host for a certain amount of time. The solution allows administrators to see when users are accessing which resources. This is an easy win for many organizations with assets in the cloud. However, an organization has no visibility into what a particular administrator is doing with that resource; the auditing is pushed off to the host that needs to be accessed.

Recently, Yelp created `dockersh`, a shell environment that is able to provide nonpersistent shell environments for users who SSH into a server [35]. This is one of the closest matches to what the system aims to do. The `dockersh` documentation does mention the issues regarding opening bastion hosts to the Internet. The system described runs a SSH daemon in the container envi-

ronment, which does allow for more separation. There is also limited discussion of good security practice in the event of a compromise. Users blindly implementing `dockersh` against the warnings of the engineers at Yelp will find themselves without any situational awareness in the event of a compromise. We mitigate these concerns by providing a "belt and suspenders" approach to security. If our container is compromised, we do have a log for a period of time that allows us to replay the attacker's movements pre-compromise.

7 Future Work

While we were able to create a system design and architecture that meets our needs and goals, there are several considerations that could enhance the security, usability, and performance of the system. Some of these items include:

- Centralized authentication is prevalent in many organizations, and it may be beneficial to leverage that as an authentication backend for the container request application.
- While it is convenient that the SudoSH utility logs all keystrokes, there are instances where this is a problem (e.g., when a user enters a password). Creating a mechanism to ignore sensitive details would be important to mitigate some insider risk.
- The container audit daemon executes rsync twice for every keystroke. We plan to implement a simple streaming data service on top of an SSH tunnel to a corresponding agent on the audit host to lower overhead for audit information.
- Migration to a Red Hat Enterprise Linux-based system would allow the use of SELinux for greater container security. Later versions of Docker will also support user namespaces, which improves the security of containers to break out even when an adversary can obtain root access inside of the container.
- A means for keeping track of the age of administrator keys, and enforcing age limits on those keys.
- Providing the SSH host key signature to the web interface (so a user could verify the key of the container they are connecting to) would be an important addition to ensure the security of the connection from a man-in-the-middle attack. This is especially relevant in Spyglass as the "trust on first use" nature of SSH host authentication provides limited benefit when the containers are re-instantiated on each session.

8 Conclusion

Given that external attackers and malicious system administrators could wreak havoc across an organization's network, it is extremely important to protect access to networks with sensitive interfaces connected to them. In this paper, we presented Spyglass, a system that utilizes auditability, nonpersistence, isolation, and multifactor authentication to protect sensitive networks. This system requires minimal change to the actual configuration of the network, provides a high security bastion, and allows an organization to securely audit their administrators' activity.

The container request application, container daemon, and audit daemon are in the process of being open sourced. The project will be updated as necessary, and pull requests from the community are welcome.

9 Acknowledgements

This paper is derived from "Demand-Provisioned Linux Containers for Private Network Access," a project completed in partial fulfillment for the degree of Master of Science in Networking and System Administration at Rochester Institute of Technology [16]. We thank Steve Stonebraker for comments and suggestions on the early design concept. Finally, we thank Thomas Moyer and Stephanie Mosely for their help reviewing this paper.

References

[1] S.M. Bellovin and W.R. Cheswick. Network firewalls. *Communications Magazine, IEEE*, 32(9):50–57, Sept 1994.

[2] Peter Bright. JPMorgan Chase hack due to missing 2-factor authentication on one server. `http://arstechnica.com/security/2014/12/jpmorgan-chase-hack-because-of-missing-2-factor-auth-on-one-server/`, Dec 2014.

[3] Pamela Brown, Jim Sciutto, Evan Perez, Jim Acosta, and Eric Bradner. Investigators think hackers stole Sony passwords. `http://www.cnn.com/2014/12/18/politics/u-s-will-respond-to-north-korea-hack/index.html?sr=tw121814nksysadminsony620pVODtopLink`, Dec 2014.

[4] Paul C. Brutch, Tasneem G. Brutch, and Udo Pooch. Indicators of UNIX Host Compromise. *;login:*, Sep 1999. `https://www.usenix.org/legacy/publications/login/1999-9/features/compromise.html`.

[5] Australian Signals Directorate. Strategies to Mitigate Targeted Cyber Intrusions. `http://www.asd.gov.au/publications/Mitigation_Strategies_2014.pdf`, Feb 2014.

[6] Dome9. Secure Access Leasing. http://www.dome9.com/overview/secure-access-leasing.

[7] Dan Farmer. Sold Down the River. http://fish2.com/ipmi/river.pdf, Jun 2014.

[8] Harriet G. Goldman. Building Secure, Resilient Architectures for Cyber Mission Assurance. Technical report, MITRE, 2010.

[9] Dan Goodin. 25-gpu cluster cracks every standard windows password in 6 hours. http://arstechnica.com/security/2012/12/25-gpu-cluster-cracks-every-standard-windows-password-in-6-hours/, Dec 2012.

[10] Dan Goodin. Aws console breach leads to demise of service with "proven" backup plan. http://arstechnica.com/security/2014/06/aws-console-breach-leads-to-demise-of-service-with-proven-backup-plan/, Jun 2014.

[11] Google. cAdvisor. https://github.com/google/cadvisor.

[12] Douglas Hanks. SudoSH. http://sourceforge.net/projects/sudosh/, 2013.

[13] Ming-Yuh Huang. Critical Information Assurance Challenges for Modern Large-Scale Infrastructures. In Vladimir Gorodetsky, Igor Kotenko, and Victor Skormin, editors, *Computer Network Security*, volume 3685 of *Lecture Notes in Computer Science*, pages 7–22. Springer Berlin Heidelberg, 2005.

[14] Y. Huang, D. Arsenault, and A Sood. Incorruptible system self-cleansing for intrusion tolerance. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, pages 4 pp.–496, April 2006.

[15] Nikolaus Huber, Marcel von Quast, Michael Hauck, and Samuel Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *CLOSER*, pages 563–573, 2011.

[16] Patrick T. Cable II. Demand-Provisioned Linux Containers for Private Network Access. Master's thesis, Rochester Institute of Technology, Dec 2014.

[17] InfluxDB. http://influxdb.com.

[18] Invincea. Freespace. http://www.invincea.com/how-it-works/containment/.

[19] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. Sub-operating Systems: A New Approach to Application Security. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 108–115, New York, NY, US, 2002. ACM.

[20] Loic Jaquemet. SSLSnoop. https://github.com/trolldbois/sslsnoop.

[21] P.G. Kelley, S. Komanduri, M.L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L.F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 523–537, May 2012.

[22] Lightweight Portable Security. http://www.spi.dod.mil/lipose.htm.

[23] Robert Love. Kernel Korner: Intro to Inotify. *Linux J.*, 2005(139):8–, November 2005.

[24] H. Okhravi, M.A. Rabe, W.G. Leonard, T.R. Hobson, D. Bigelow, and W.W. Streilein. Survey of Cyber Moving Targets. Technical report, MIT Lincoln Laboratory, Jul 2013.

[25] OpenVPN. http://www.openvpn.net.

[26] B. Parno, J.M. McCune, D. Wendlandt, D.G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 154–169, May 2009.

[27] Mike Pope. Securely connect to linux instances running in a private amazon vpc. http://blogs.aws.amazon.com/security/post/Tx3N8GFK85UN1G6/Securely-connect-to-Linux-instances-running-in-a-private-Amazon-VPC, Sept 2014.

[28] Pythian. Adminiscope. http://www.pythian.com/products/adminiscope/.

[29] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of os-level virtualization technologies. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Secure IT Systems*, volume 8788 of *Lecture Notes in Computer Science*, pages 77–93. Springer International Publishing, 2014.

[30] SANS Institute. Critical Security Controls. http://www.sans.org/critical-security-controls.

[31] Andy Sykes. Ganger. https://github.com/forward3d/ganger.

[32] Ashish Thakwani. Process-level Isolation using Virtualization. Master's thesis, North Carolina State University, Jan 2010.

[33] David A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Master's thesis, University of California, Berkeley, 1999.

[34] Zachary Wikholm. CARISIRT: Yet Another BMC Vulnerability (And some added extras). http://blog.cari.net/carisirt-yet-another-bmc-vulnerability-and-some-added-extras/, 2014.

[35] Yelp. HACK209 - dockersh. http://engineeringblog.yelp.com/2014/08/hack209-dockersh.html.

[36] Yubico. Yubikey standard. http://www.yubico.com/products/yubikey-hardware/yubikey/.

[37] Elizabeth Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls*. O'Reilly Media, second edition, 2000.