Image import and SSH security in Emulab

Srikanth Raju

July 15th 2013

Abstract

Cloud providers typically provide compute facilities for their users. It would be very convenient if users' could import and export their machines with much greater ease to other cloud providers or to their desktops or local machines. Here, we talk about how we've enabled users to easily do this with Emulab. Specifically, we've implemented image import mechanisms to import machines from cloud providers like OpenStack and EC2 and also any other general regular Linux machine.

We also discuss how we've implemented a SSH security system in Emulab that helps alleviate some of the security that might be brought in by such imported machines. A man in the middle attack is used in a positive way to enforce specific policies onto SSH connections. By doing this, we can apply additional constraints over incoming SSH connections that may not be enforced by the SSH server.

1 Enabling external image usage for Emulab

Infrastructure-as-a-Service providers typically have specialized environments that require specific operating system adjustments to run them. Portability between these various cloud providers is usually lacking. Emulab is a network testbed is designed for users to test their applications with fine grained control for network parameters. With the growing prevalence of cloud platforms like Amazons' Elastic Cloud Compute(EC2)[1], users of these services would like to bring in their entire stacks running on EC2 and do network specific tests on Emulab. Here, we discuss how we have integrated support for machines running on EC2 and OpenStack in Emulab and how users may import them easily into Emulab, so that they can run finer tests.

1.1 Background

Emulab[11] allows you to load various images onto a physical or virtual machine and include them in a network experiment. This feature is quite similar to what various cloud providers give in relation to their compute facilities. However, Emulab provides various additional features, such as the ability to manipulate network topologies and have reproducible network effects.

It would be very convenient to allow users to bring in their own images from other cloud providers or even their own desktop machines. The idea is to make such a process very simple and to be portable with other cloud providers. Emulab also allows users to run images on both physical machines and virtual ones. However, we plan to allow these imported machines to run only in virtual mode on Xen. This is because for running on physical machines, we require the machine to automatically configure all its' own network interfaces(Emulab machines have more than one), run all the network traffic daemons and install user accounts. However, by allowing only virtual machines, most of those Emulab specific aspects can be controlled by the Xen host.

EC2 and OpenStack typically have a clientside that configure specific things about a machine when they boot up. These clientside programs use an instance metadata mechanism that allows images to get information about their configuration. This is similar to a Testbed Master Control Daemon/Client mechanism[6] that is present in Emulab and is used to configure the Emulab clientside on Emulab images. The most important things that are configured through the EC2 metadata API are the SSH keys of the machine and network information. We have implemented this instance metadata API as a server to allow EC2 images to be able to configure themselves.

1.2 Image importing

Linux AMIs in Amazons' EC2 service typically run on a Xen Hypervisor. While Amazon can run both Linux and Windows systems, we are mainly interested in Linux at this moment. The images for Amazon are packages into Amazon Machine Images. These AMIs are signed by a special certificate that belongs to the owner of the image or Amazon. Also, a user can only download their AMI if they own the base image. However, if a user has a running machine, we can create an image of the running volume and package it for use with Emulab. The situation is almost the same with OpenStack.

We have implemented image importing in Emulab. The general process can be seen in figure 1. We provide a web form for users in Emulab to provide an publicly accessible hostname that corresponds with the machine they would like to import. The ops machine on Emulab connects to the target machine. The `grub.cfg` is parsed so that the kernel and ramdisk can be pulled out. This is important because we want to boot these machines on a Xen hypervisor which requires the initial ramdisk and kernel to be given to it to load the operating system. A large enough file is then `dd`ed on the target machine and a ext3 filesystem is `mkfs`ed onto it. The root filesystem then gets `rsync`ed into the file. This requires quite a bit of free space to be left on the machine. However, it is assuaged by the fact that it is possible to mount additional temporary block storage on most cloud providers today. We then compress the filesystem into a tarzip and copy this back into our ops machine. Most images are between 500MB to 2 GB. It may be noted that while it can be quite cumbersome to transfer this over the network, it is the most efficient, because it is the most direct way to transfer this data.

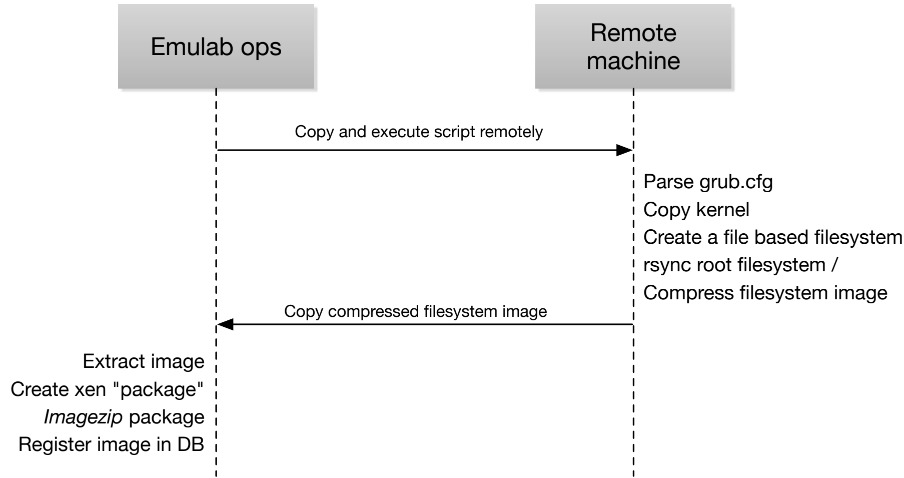

Figure 1: Image import into Emulab

Emulab uses `imagezip`[8] for compressing and storing images. Imagezip compresses specifically using filesystem knowledge and thus is very efficient. The kernel, ramdisk and filesystem are extracted and tarzipped up to create an image that can be used by emulab. When a virtual machine is loaded with the image that was created, the kernel and ramdisk are pushed onto Xen along with the filesystem image being loaded onto a LVM partition.

1.3 Instance Metadata and Clientside self configuration

EC2 uses an instance metadata service[2] to provide running systems with data that they can use to initialize local services. OpenStack also supports the same service[5]. Images that are designed for EC2 or OpenStack access this service and use them for self initialization on boot, typically only for the first boot. Initialization scripts that use this service typically configure the network, access to the machine or some other configuration that can't be burnt into the image. This is the method that images use to pull in SSH public keys for the user and thus is very important to support.

The instance metadata service runs on a special link-local address `169.254.169.254`. This address is a link-local address, so they need to specifically configured to the correct service within our network. The metadata service runs on port 80 and uses the HTTP protocol. Thus, it is possible to access this metadata with simple HTTP requests using tools like wget or curl. The data is accessed from the base URL `http://169.254.169.254/latest/meta-data`. The most important metadata that is served are the SSH public keys of the

user. For this specific reason, we have a specific metadata server running that provides the running machines with this data. Other important metadata that are useful for the image are the local hostname of the machine, IP address and the instance ID of the machine. We have implemented this as a Python server using data directly from Emulabs' database. The list of metadata that we have implemented support for are in table 1. The ones that we have not supported are extremely EC2 specific.

Data	Description
ami-id	The Image ID used to launch the instance.
mac	The instances' MAC address.
instance-id	The ID of this instance.
public-hostname	The public hostname of the instance.
public-ipv4	The public IP address.
public-keys/0/openssh-key	Public keys of users

Table 1: Supported Instance Metadata in Emulab

Since we need to make `169.254.169.254` accessible to the machines that require this instance metadata, we use netfilter on the Xen host machines to redirect packets bound for that address. Instead, we run a metadata server on some port on the Emulab boss machine. This metadata server was written in Python and pulls out the same data that is pulled out from the Emulab database. Simple `iptables` rules can be used to operate on the bridge interface on the Xen host to perform a redirect through the DNAT target[3] while routing the packet.

2 SSH Security

2.1 Introduction

It is difficult for infrastructure-as-a-service (IaaS) providers to maintain security in the face of users' poorly configured virtual machines. The users of clouds and network testbeds create virtual machines that are exposed to attacks from the Internet, and poorly configured VMs are quickly compromised. SSH login attacks are very common and machines with very common passwords are easily broken. If a user has a weakly configured system, there isn't much a provider can do.

The problem is amplified by the fact that in Emulab, we have much stronger trust within the network between the machines. This is because till now, most of the machine operating systems have been created by Emulab administrators and configured to be secure. We expect machines to be mostly well behaved. Allowing users to import their machines means that we have very little control over them and we have no means of checking if a users' account is poorly configured. The Utah Emulab has had atleast two attacks where users' machines

have been compromised because of such security problems. This is one of the motivating factors for us to solve the probem.

Currently, cloud vendors such as EC2 typically do very little about poorly configured SSH servers. Amazon EC2 provides guidelines to configure SSH servers and asks users to use public keys instead of passwords. If a VM is compromised and are being misused for activities like port scanning, Amazon typically locks up the server and notifies the owner. It is important to note that Amazon probably has firewalls which are configured to completely reject any connections from known bad hosts. However, a cursory look at the SSH daemon logs on a EC2 machine that was up for just under 24 hours shows that people trying to break into SSH are not for the lesser. The machine received 65 login attempts from four different IPs. This shows that attacks happen even on enterprise secured networks.

We have implemented a friendly SSH man in the middle for Emulab, an IaaS network testbed. The challenge in this is that SSH is designed to be unbreakable, so it is nearly impossible to snoop on the data being transmitted on an SSH session between two systems. It is also important to maintain transparency so that legitimate users do not have to jump through hoops to connect to their machines. Our goal is to design a system such that we are able to protect Emulab's users while not introducing newer and more subtle security issues.

2.2 Methodology

Secure Shell (SSH) is a cryptographic network protocol for secure data communication and remote login. It is the primary method of logging in to remote machines. It is the prime method to connect to remote machines.

There are two versions of the protocol. The first version is obsolete. SSH v1 is susceptible to injection attacks due to weak data integrity protection(CRC-32)[12]. It also allows for only one channel at a time, one authentication per session and supports very limited encryption and integrity algorithms. The newer SSH v2[16] protocol is the one that is widely used and is much more secure.

While it is possible for server administrators to configure popular SSH server applications with conditions such as "no root authentication", it is much more difficult to do so on virtual machines that any user can import with their custom operating systems because administrators do not have direct access to these machines. One of the ways of doing this is to modify the guest filesystem. So, such a method requires us not only to be able to read a variety of filesystems, but also to support a variety of server versions. It also enables the user to change it back if we do not continuously monitor the filesystem. More importantly, it modifies something that is supposed to belong to the user - their data.

The method that we chose to implement involves adding a layer in between the client and the server. The goal of doing a man in the middle in traditionally to attack a server and to be able to masquerade an SSH session and to ultimately gain control of the machines. The malicious users can these use machines for their nefarious purposes. However, it is possible to use the same technique to

help increase security of these systems. A variety of security policies can be forced upon SSH servers. Some of our goals with such a system is to disallow weak passwords, connections from specific servers or users and to lock down SSH login attacks.

2.3 The SSH Protocol

The SSH protocol v2[16] is a layered protocol which supports multiple sessions and channels for mechanisms such as port forwarding and file transfer. The main protocol has three different layers - the Transport Protocol, Authentication Protocol and the Connection Protocol. The Transport layer provides server authentication, confidentiality, data integrity and compression. The authentication layer mainly does password and publickey authentication and interfaces with the OS authentication modules like PAM. The connection layer provides channels for user sessions and other channels for forwarding.

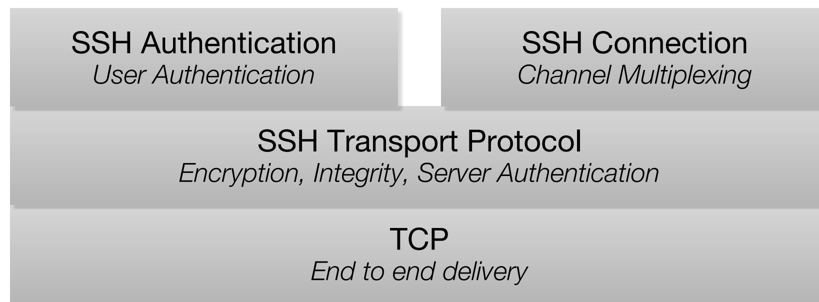

Figure 2: TCP layer

The SSH Transport protocol[15] negotiates a shared secret using Diffie Hellman key exchange. It then generates encryption keys for transferring data using this shared secret. During the initialization of the protocol, the client also requests for the servers' host key. This layer also generates a message authentication code of the unencrypted data, so that the data can be verified. The entire process is protected by the server signing with its' host key, and sending the signature to the client so the servers' public key can be authenticated. The Authentication protocol[13] can be used to support mainly two authentication methods - password and public-key. The SSH protocol also supports host based authentication, however, this is hardly ever used, so we have focussed on the first two methods. The transport is disconnected is the authentication fails. We will discuss how authentication takes place and how it affects our implementation in section 2.5 and section 2.6. The Connection Protocol[14] acts as the channel for carrying data to different processes. There are various channels that run on the connection protocol. The session channel, the x11 forwarding channel and the

port forwarding channels. The connection handles demultiplexing of the stream data to these separate channels. The session channel is what typically carries the terminal data.

2.4 The Man in the Middle

It is possible to perform man in the middle attacks for SSH. This is however not fully transparent to the client user because the SSH servers have to present their host key during the key exchange during the initiation of the Transport layer. This means, that it is impossible for a man in the middle to completely pretend to be a different SSH server, unless, ofcourse, they have the private host key of the SSH server in question because the server has to sign a shared hash with their private key. The first time a SSH client connects to a server, it usually stores the host key in a file. This way, a client can detect when a key changes and indicate to the user that they might be getting attacked. This isn't much of a problem for our use case, because we are not a threat to the user. Instead, we can publish our public key, so that users can trust the gateway SSH servers.

The general method of doing the man in middle involes performing a bridge between the client and the actual server. For every request the middle server receives, it performs a corresponding request at the target server. This should be performed at the layers above the transport layer. This is because the transport layer is protected by encrpytion using a session key. Since the session key is generated by diffie hellman, it is impossible to force the same key over the bridged session. Thus, it would be impossible to see inside the messages and bridge them onto the target at the same time.

The goal is to enforce policy and we can use this man in the middle to reject connections which we deem are unsafe or questionable. The specific policies that we can enforce will be discussed in section 2.7.

One of the important things to discuss is how the SSH connection is actually routed into the middle server instead of actually heading to the actual server. Attacking agents use various methods such as ARP spoofing and DNS spoofing to make the client connect to wrong machines. In our situation, the problem is handled much more simply. Since our imported machines run on a virtual machine, they all run on a single machine with a bridged network interface. In Emulab, the Xen host machine runs a Ubuntu server distribution running Linux 3.2.46. The network stack on the host system can easily be accessed using Netfilter, which provides a set of callbacks to the network stack. The guest operating systems' network interface are typically configured to get bridged over to the bridge on the host side. Using `iptables`, we can easily add a redirection rule[4] that redirects all connections bound to the SSH port to the port on the same machine where the man in the middle is running.

The other important consideration is that of which machine to perform the man in the middle on. While the normal methodology described above works well for one server, it is possible to do the man in the middle for multiple servers. The server to connect to can be encoded via different methods. One method is through the username. For example, the client should connect to the machine

with username userfoo:node0. The node0 in the username encodes the target machine that needs to be connected to. We call this problem of determining targets the demux and the username based demux is one of the demuxes we have implemented. The standard simple demux is called the port demux because the target is determined by port the man in the middle is listening on. A demux can potentially work in many ways - using keyboard interactive authentication to ask for the target, setting up the target based on the users' key or connecting users to their respective machines based on some registration. This kind of feature is useful for special servers or in cases where there is a gateway machine and the administrator wants to perform specialized checks on incoming users.

The implementation of this system is done in Java. We use a library called `sshtools` which provides the mechanism for talking to servers. This provides the base layer for implementing the system. `typesafe-config` and `akuma` provide the configuration file parsing and daemonization respectively.

2.5 Password Authentication

The password based attack is fairly straightforward. This kind of attack is well known[7, 10] and can be easily performed. The authentication token - the password is transmitted in plain text and is thus vulnerable either to steal or to authenticate a malicious user. We can easily mirror every message coming to the man in the middle server (MITM server) to the equivalent client request to the actual server(the target). The process is shown in figure 3. When the transport connection comes in, we initiate a separate transport with the server. When the client requests password authentication, we receive the username and password in the SSH_MSG_USERAUTH_REQUEST message. We initiate a similar message to the server and if the target server succeeds, then we reply with a success message, otherwise we reply with a failure. Once the user is authenticated, any of their connection channels can immediately be opened and we send requests for the same channels to the server. For all the channel data that is received, it is simply re-encrypted and sent over the target connection and vice versa. During each phase of the process, we can apply our own filteration process to apply any additional policies that we want to. Thus, we can simply reject all authentication for user "root" even if such an authentication would pass at the target server.

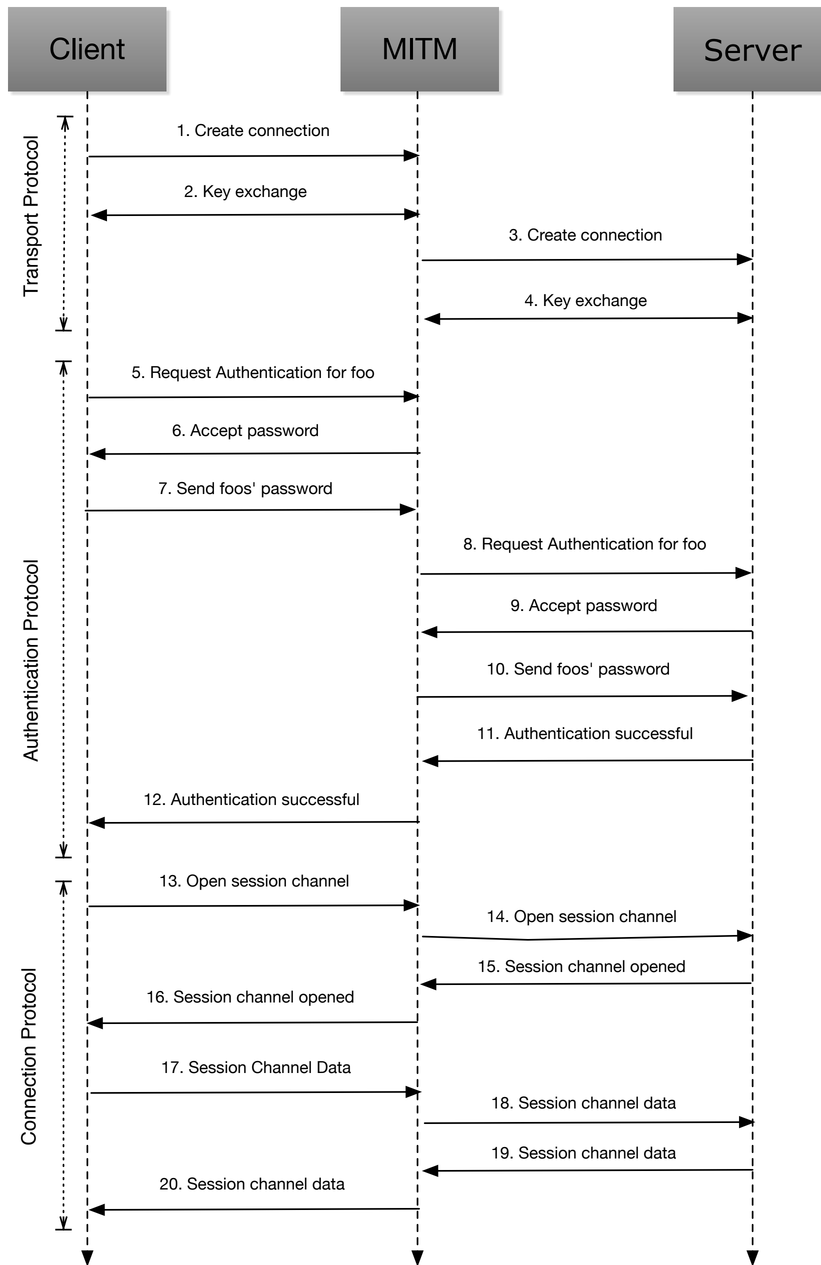

Figure 3: The man in the middle for passwords

2.6 Public Key

The above technique however does not work for public key authentication. Public key authentication is resistant to man in the middle attacks because during authentication, the client will sign a packet containing the session with their private key. In order to authenticate with the actual server, the middle man will have to sign a similar packet with the private key of the user. Ofcourse, this is usually unobtainable to attackers. Even for our model where the goal is to improve security, it would be absolutely ludicrous to request the user their private key. One possibility is that we could create a separate key and use that to authenticate between the middle man and the imported machine. This creates an additional overhead for the user and it means that our server can connect to the users' server as well, which might not be comfortable.

The alternative is to use an inbuilt protocol mechanism - agent forwarding[9] to achieve our goals. The process can be seen in figure 4. Once the connection is received at the gateway server, we proceed until we receive an authentication request. When we determine that the request is for public key authentication, we return an authentication successful message. The client then proceeds to make requests for whichever services it requires. However, we don't process any requests, but instead we queue all of them and wait for an agent-auth-req request on the session channel. This is the request from the client machine that sets up agent forwarding. As soon as this message arrives, we set up the agent forwarding and try to connect to the real server. The mechanism forwards the signing request to the actual client, which does so on behalf of the man in the middle. Again, the same technique can be used by attackers to hijack user sessions or to authenticate as the user to different machines during the session. Once we ensure that the user has been authenticated, we can forward all the queued messages. In the event that the user fails to authenticate, we disconnect from the client.

The important thing to note in the public key method is that even though the client has been told it is authenticated, it actually isn't. Also, we open the session channel even though we have no real access to a terminal session on any machine. Instead we wait for the user to agent to signal that they have forwarded their agent and queue all other messages for all channels. There is also a timeout which disconnects the whole connection if the user hasn't enabled agent forwarding. The general architecture can be easily visualized in figure 5. The dotted lines represent messages and the double lines indicate that the messages between those channels are passes as-is.

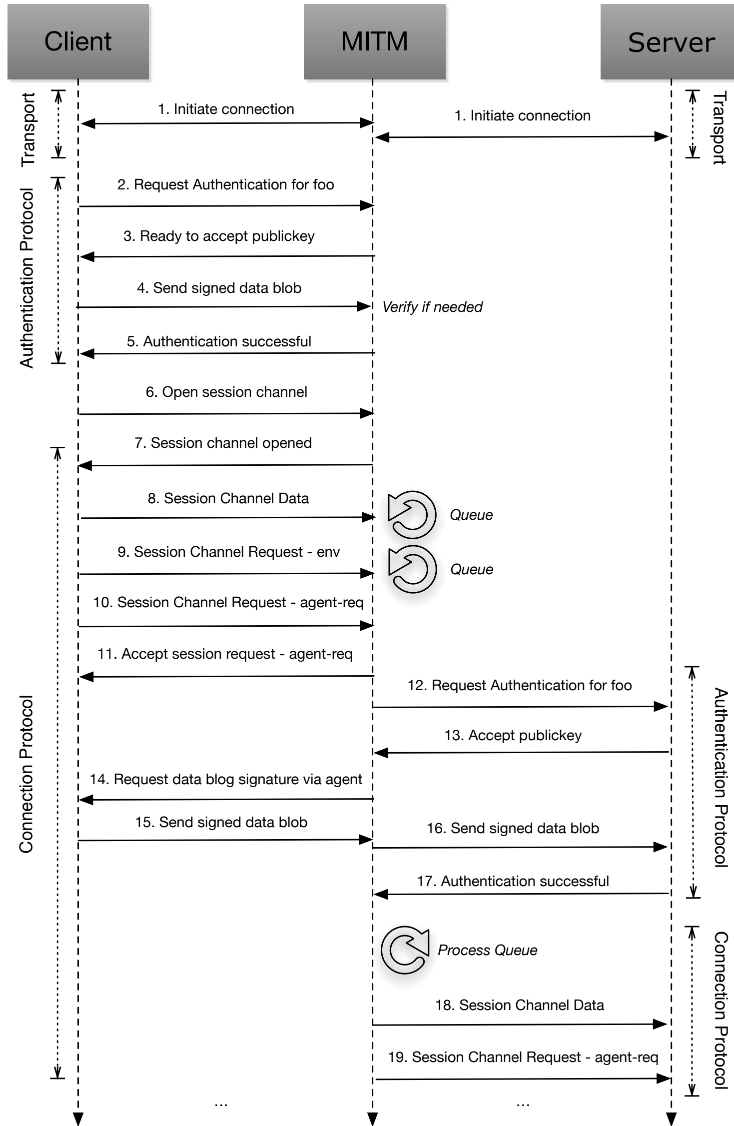

Figure 4: The man in the middle for publickey

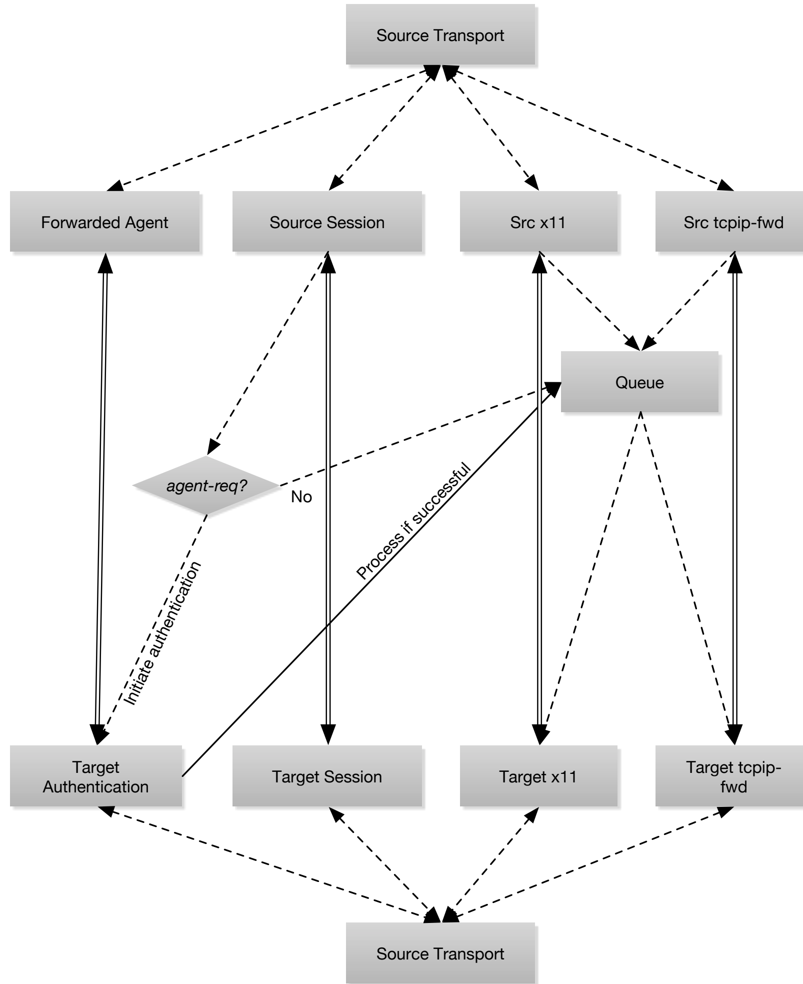

Figure 5: Queuing messages and bridging architecture

Again, with this authentication mode, it is possible to do all the additional policy we would want to. We could enforce all the connection level tests like blocking specific hosts, but also test for specific users.

The security of the entire system still hinges on the user checking the host key correctly. If the user accepts any host key and connects to some machine other than the target machine or the MITM server, then they are insecure no matter what. Our implementation does not do anything to alleviate this problem with SSH. However, the MITM servers public key is published for everyone, so the user still has to take the usual tedious steps to verify that the key matches.

The man in the middle system fixes only the problems associated with poor configuration and is succesful in that.

2.7 Policy enforcement modes

The main point of doing the man in middle was to force policy onto the servers from outside. All of these configuration options are available from a simple text configuration file that is provided to the MITM server and have allow or disallow modes.

- Authentication - Allow only password, publickey or both types of authentication.
- Password - Allow or reject only certain passwords, for example - empty passwords or the password "password".
- Publickey - Allow or reject certain public keys or have stronger and additional public key verification.
- Users - Allow or reject certain users from logging in. Useful for disallowing root logins.
- IPs/Hosts - Allow or reject known bad hosts.
- Brute Force Detection - Block machines from trying to connect more than a certain amount of times per minute.
- Channels - Allow or reject TCP forwarding or X11 forwarding

Each option is considered to be a regular expression, so it's possible to write a general expression for rejecting cases or accepting only in a few cases.

3 Conclusion

We've seen how we have added support to import images into Emulab from remote machines with a simple process. This also includes the metadata server to perform self initialization for images coming from EC2 and OpenStack. By doing this, we've greatly increased the portability of images with Emulab. We have also implemented a configurable system to improve SSH security of machines running inside Emulab by applying additional checking of constraints on SSH connections. This system can be used to ensure that SSH servers use strong policies that greatly reduce the possibility of attack. Overall, with these two systems, we have improved the portability of virtual machines and the security configuration of the same.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1059440. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amazon elastic compute cloud (amazon ec2). URL: http://aws.amazon.com/ec2/.

[2] Instance metadata and user data. URL: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AESDG-chapter-instancedata.html.

[3] iptables dnat target. URL: http://www.linuxtopia.org/Linux_Firewall_iptables/x4013.html.

[4] iptables redirect target. URL: http://www.linuxtopia.org/Linux_Firewall_iptables/x4508.html.

[5] Metadata service. URL: http://docs.openstack.org/trunk/openstack-compute/admin/content/metadata-service.html.

[6] Testbed master control daemon/client. URL: http://www.emulab.net/doc/docwrapper.php3?docname=tmcd.html.

[7] Julian Beling. Conducting ssh man in the middle attacks with sshmitm. URL: http://www.giac.org/paper/gsec/2034/conducting-ssh-man-middle-attacks-sshmitm/103515.

[8] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, scalable disk imaging with frisbee. pages 283–296, San Antonio, TX, June 2003.

[9] D. Moffat. Ssh agent forwarding, December 2001.

[10] Alberto Ornaghi and Marco Valleri. Man in the middle attacks demos, 2003. URL: http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-ornaghi-valleri.pdf.

[11] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. pages 255–270, Boston, MA, December 2002.

[12] R. Michael Williams. Exploiting the ssh crc32 compensation attack detector vulnerability, 2001. URL: http://pen-testing.sans.org/resources/papers/gcih/exploiting-ssh-crc32-compensation-attack-detector-vulnerability-103026.

[13] T. Ylonen and Ed C. Lonvick. The secure shell (ssh) authentication protocol, January 2006.

[14] T. Ylonen and Ed. C. Lonvick. The secure shell (ssh) connection protocol, January 2006.

[15] T. Ylonen and Ed. C. Lonvick. The secure shell (ssh) transport layer protocol, January 2006.

[16] Tatu Ylonen and Chris Lonvick. The secure shell (ssh) protocol architecture. 2006.