Hedwig: Secure dialup service with strong per-user sandbox

Erin Main, Lizhou Sha
{ermain, slz}@mit.edu
15 May 2017

Abstract

Hedwig is a secure dialup service with VM-based per-user sandbox, intended to replace the current Athena dialup servers at MIT. Compared to a traditional multi-user Unix system providing shell access, *Hedwig* provides a more secure and stable environment for its users by preventing a wide class of local privilege-escalation and denial-of-service attacks. At the center of *Hedwig* is an SSH proxy that accepts incoming connections, spins up a new container-VM for each user using rkt, and proxies user requests to the container. Currently, *Hedwig* exists as a proof of concept that supports login of manually configured users into a stock Ubuntu system. Further work is required to enable remote users as well as all the bells and whistles that make up an Athena dialup server.

Introduction

At MIT, Information Systems and Technology (IS&T) provides account holders SSH access to the Athena dialup servers, which are Linux servers bas*ed* on the specialized Debathena distribution. These servers give users a remotely-accessible option for doing tasks that require the Athena computing environment. For instance, users can attach Athena lockers and store data in their networked home directories via AFS (Andrew File System). The Athena dialup servers are extremely popular, and provides a practical alternative to users installing the Debathena distribution on their own computers. However, as with any multi-user Unix system, there are several security shortcomings of the Athena dialup servers:

- Lack of user-level sandbox. Unix-based user privilege separation and AFS-based file system access control form the bulk of isolation between users of the dialups. Users share the same local filesystem and process namespace.
- Lack of resource constraints. There are no per-user resource constraints, so one user can "resource-hog" to the detriment of all other users.
- AFS instabilities. Due to the lack of isolation between kernel modules, the buggy AFS kernel module can cause an entire dialup to go down when it crashes. This previously occurred on several of the Student Information Processing Board (SIPB) servers.

In light of these issues, we have designed *Hedwig*, a dialup service that sandboxes each user in their own lightweight VM. In theory, this system could prevent an attacker with login privilege on the dialup from attacking another user by exploiting local privilege-escalation vulnerabilities. Imposing resource limits on those VMs would prevent a wide class of malicious or accidental denial-of-service attacks that affect the current Athena dialups.

For this project, we have chosen to focus on the core of the idea (spinning up simple VM-based containers and forwarding SSH connections into them) instead of trying to implement the entire system with the Debathena environment.

Design and Implementation

Technologies Used

We built our system out of a few existing concepts and libraries.

VMs + Containers

Containers allow for multiple userspaces to run in a single userspace. Each container can be isolated by the host kernel into their own namespace (similar in concept to chroot), and can also be resource-limited using control groups or cgroups. Containers are fast to startup, since they are just userspace programs.

However, since a proper Athena dialup must run the AFS kernel module and make AFS available to containers, a misbehaving container has the ability to cause a host kernel panic, bringing the entire machine down with it. In the case of the Athena dialup servers, the somewhat mercurial AFS kernel module is a source of frustration for system administrators and users alike.

Due to the infamous AFS kernel module, we want to provide hardware-backed isolation for the containers, preferably at the virtual machine level. Virtual machines (VMs) have great isolation properties; if a guest kernel crashes, it will not affect other guests or the host. However, virtual machines also traditionally come with a fairly long boot time, as booting a virtual machine requires going through all the same boot steps as on a normal, bare metal machine.

For performance reasons, we use the Intel Clear Containers technology, which combines the fast boot time and easy management of containers with the isolation properties of VMs.[1] Clear Linux is a stripped-down Linux kernel and userspace, only containing what is needed to boot on a virtualized system. An optimized KVM/QEMU (hypervisor) can then launch the Clear Linux kernel, which in turn can run traditional client images.

For the container engine, *Hedwig* uses rkt, with a container launch system based on the Intel Clear Containers model in lieu of the default Linux seccomp and cgroup-based containers.[2] rkt was chosen over Docker (another popular container engine) because rkt is a more security-minded engine.

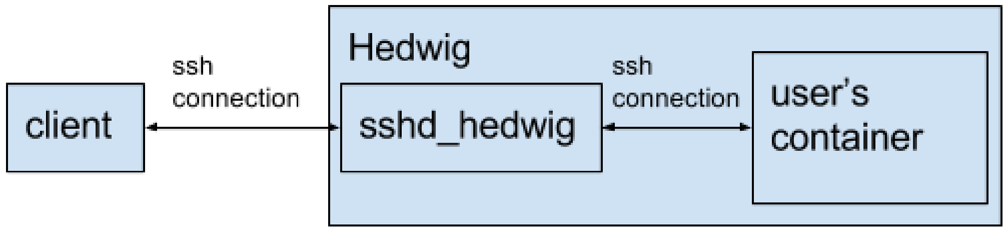
SSH

Secure Shell (SSH) is a cryptographic network protocol which provides a secure channel for doing operations over an insecure network.[3] Typical uses include logging into a server remotely, and remotely executing commands on a machine. An established SSH connection contains several elements: a set of *channels*, which transfer data; a set of *channel requests*, which relay out-of-band information about the channels (such as a pty-request or window size change); and *global requests* (such as a request to start TCP port forwarding over the SSH connection).

We use public key-based user authentication in *Hedwig*. The OpenSSH utility ssh-keygen generates a public/private keypair; the public key is stored on the server to be ssh'd into. Thereafter, the private key can be used to connect to that server. Clients can also verify they are connecting to the right server by (out-of-band) saving the server's public key to a known_hosts file. On connection to the server, the client checks that the server is presenting the same public key as is saved in the known_hosts file.

For *Hedwig*, we chose to use an existing SSH implementation provided by the Go authors, which gives primitives for building an SSH client and a host in Go.[4] Go is also an ideal language to use due to its excellent primitives for concurrency. Go's goroutine construct, which provides a lightweight threading interface, provide us a way to handle large amounts of potentially blocking requests asynchronously and safely.

System Components



In *Hedwig*, there are three components that need to communicate with each other: client, which is just an ordinary SSH client run by the user; container, which is a lightweight VM that runs an SSH server that processes client requests; and host, which authenticates and accepts client connections, launches a new container to handle client requests, and sets up an SSH tunnel between the client and the container.

Container

For the purposes of developing our proof of concept, we have chosen the standard Ubuntu 16.04 (Xenial Xerus) Docker image as our container image, as it saves a significant amount of time building during development.[5]

To generate our custom *Hedwig* container image from the Xenial base image, we used the tool acbuild, which allows one to create container images that are compliant with the App Container specification.[6] Our build script installs openssh-server and configures the container to use rkt-mounted keys for client / host verification between the container and the host. We also inserted a script that runs on container startup; the script creates a user using “useradd”, creates the temporary directory /var/run/sshd, and then launches sshd in the foreground. A strange quirk of working with this containerized-ssh is that if sshd is launched as a traditional Unix forked daemon (as is typical on most machines), then the container will immediately exit. We note that the sshd we run in the container is the standard sshd installed by openssh-server.

Host

At the present, the host is a machine with hardware virtualization running Ubuntu 16.04. For container management, we use rkt, which also provides us the ability for service discovery.

sshd_hedwig (our SSH proxy -- the source file is sshd_hedwig.go) is a custom program written using the Go language’s SSH library. On startup, it loads the Hedwig host private key and a list of public keys from authorized_users into memory; then it listens on port 2222 for a user connection. When a user connects on port 2222, the resulting TCP connection is accepted, then passed to the SSH library to perform authentication using public-key crypto. The SSH library then passes back three objects: the encrypted client connection, a Go channel containing new SSH channels, and another Go channel containing new SSH global requests. We delay handling these connections until the container is successfully set up.

Upon authentication of a user, sshd_hedwig starts a container by passing several arguments to rkt: the username of the connecting user; the mount point for key material; and the method to run the container (KVM). The key material includes a container host private key, and a public key for sshd_hedwig, both used to connect to the container. Once the container is successfully started, sshd_hedwig waits some time to allow the sshd within the container to start up, then attempts to establish an SSH connection into the container, using the aforementioned key material.

At this point in execution, sshd_hedwig contains a handle for an SSH connection back to the client, as well as a handle for the SSH connection into the container. The final step is to relay requests and data back and forth between the two connections. The general approach for both channel and global requests (the latter is called “connRequests” in the code) is to look at the type of request coming in on one connection, then manually make the same request on the destination connection; when a reply is received from the destination connection, sshd_hedwig then replies to the origin connection. For channels, if sshd_hedwig gets a request from the client to open a new channel, then sshd_hedwig requests a new channel from the container. If the request succeeds, then sshd_hedwig connects the two channels. This connection is accomplished by reading data from one channel and writing to the other channel; this occurs in both directions, so that data can be transferred back-and-forth between the client and the container.

When a client closes the SSH connection, this causes the goroutines handling the channel data to return. This triggers shutdown and clean-up of the container using the defer statement in Go.

Security Considerations

Hedwig is a multi-tenant system. Therefore, our threat model mostly involves protecting users of the system from other potentially malicious users. A client must assume that *Hedwig* is a trustworthy system, since *Hedwig* is forwarding data from the client's connection to the container. Therefore, it is paramount that clients verify that they are connecting to a trustworthy instance of *Hedwig* by verifying the host public key fingerprints out-of-band, consistent with the normal SSH trust model, however.

We have chosen to make the connection between sshd_hedwig and the containers also happen over SSH, making sure that sshd_hedwig and the container SSH server mutually authenticate. Although this layer of SSH is technically not necessary, as the container's virtual Ethernet interface is point-to-point connected to the host, we want to defend against hypothetical active attackers with remote SSH access to *Hedwig* who exploit vulnerabilities in rkt's network management, such as by ARP spoofing, etc.

Since we are providing a dialup service, the usual considerations for network security also applies. Thankfully, the mature SSH protocol guarantees our security against network snoopers and MITM attackers. On the host, we used the existing Go SSH library in order to avoid the pitfall of implementing our own cryptographic primitives. Presumably, the clients will use the OpenSSH client to talk to *Hedwig*, which also runs the OpenSSH server inside of the containers, so we also rely on the OpenSSH implementation to be secure.

Conclusion and Future Work

We have built a proof of concept for *Hedwig*, a dialup server that can provide full isolation between logged-in users. The current system is able to authenticate a client, start a container, and then forward the SSH session and requests from the client to the container. However, there are many tasks to be done before the system is deployable for use by the MIT community. A subset of these include:

- Enable long-running sessions
- Allow non-local users in containers
- Kerberos-based authentication for SSH proxy
- Optimize/augment the default Docker image further to give users a fuller Athena dialup environment

Acknowledgements

The authors would like to thank Alexander Chernyakhovsky for technical assistance and inspirations that led to *Hedwig*, as well as Professor Nickolai Zeldovich for suggestions that fleshed out *Hedwig*'s security model.

References

1) Clear Linux project. "Intel Clear Containers." *Intel Open Source*. https://clearlinux.org/documentation/clear-containers/clear-containers.html#architecture-overview. Accessed May 14th, 2017.
2) CoreOS team. "Running rkt with KVM stage1." *rkt documentation.* https://coreos.com/rkt/docs/latest/running-kvm-stage1.html. accessed May 14th, 2017.
3) T Ylonen. "RFC 4253: The Secure Shell (SSH) Transport Layer Protocol." *Internet Engineering Task Force*. https://tools.ietf.org/html/rfc4253. Published January 2006, accessed May 14th, 2017.
4) The Go authors. "package ssh." Godoc.org. https://godoc.org/golang.org/x/crypto/ssh. Accessed May 14th, 2017.
5) The Docker team. "ubuntu." *hub.docker.com*. https://hub.docker.com/_/ubuntu/. Accessed May 14th, 2017.
6) The containers team. "acbuild: another build tool for container images." *Github.* https://github.com/containers/build/releases.Accessed May 14th, 2017.