2016

Guide to User Data Security

By Brian Pontarelli

Table of Contents

1 Overview

“LinkedIn hack releases 167 million accounts.”

“Milwaukee Bucks organization accidentally leaks player financials onto the Internet.”

“4.8 Million V-Tech customer’s data stolen.”

These headlines scream: software security is a big deal.

While thinking about software security often creates that sinking, pit-like feeling in your stomach, it pales in comparison to the nausea that strikes when the CEO is on the news explaining that the software system you built leaked private information about users.

There’s no industry, no organization and no classification of software that is immune to the predatory antics of hackers. Personal information, corporate data, even high-profile social media accounts are under constant attack. Any server system accessible from the Internet is not just a potential target, but an actual target. When Microsoft first starting working on their Windows 2000 software system they wanted to see how well it would resist attack. To test this, they put a few servers onto the network and waited. Within hours the attacks began. We conducted the same experiment in 2016 and it took less than 60 minutes for the first brute force attack to come in from overseas.

So as I mentioned before, software security is a big deal.

Security is complex and for many software professionals an incredibly intimidating task that is easy to just ignore as a problem “outside the scope of requirements”. However, this is no longer a viable option. Security has moved from a “we’ll get to that eventually” feature to a priority. Security must be skillfully factored into designs, projects and timelines.

This guide is for the software developer, architect or system administrator who doesn’t want to spend a lifetime wading through cryptographic algorithms and complicated explanations of arcane system administration topics to tackle software security. We are a software development company and we have taken everything we know (and have learned through the years) about server and application security and distilled it into this simple yet detailed guide. This is not the sum of all things that could be or have been said about software security, but if you implement each of the concepts below your user data will be highly secure.

There are two parts to the guide: Server Security and Application Security. We don’t see one as more important than the other, so we strongly encourage readers to digest both sections with equal attention. We provide a series of guidelines and steps to harden the servers/applications and protect against direct attacks. This isn’t a “Why” kind of guide; this is a “How” guide. We trust that those interested in the “Why” will find plenty of other resources to satisfy their curiosity. For those of you interested in the “How”, let’s get started.

2 TL;DR

If you would rather not read our entire security guide (though we suggest you do), check out the Github project created around this guide. This project contains a set of scripts you can execute from your local computer to secure a remote server. The scripts perform most of the steps listed in this guide. The Github project also contains full configuration files based on the configuration detailed below that are deployed to the server by the scripts.

NOTE: That is only half of the equation, so you will still want to read the Application Security section of the guide.

Find our Github project here: https://github.com/inversoft/2016-security-scripts

3 The App and Contest

We wrote a simple application in order to ensure everything in this guide worked correctly. Our application is a simple to-do manager. Users can register for an account and log into the application. Once logged in, users can manage their to-do list. Our application uses Node.js, Express and Sequelize for the server-side, Ember.js for the front-end MVC, and MySQL and Passport for the databases. Passport is used to store user data and MySQL is used to store the to-dos.

We've open sourced all the code and deployed our application to a couple of Linode servers. We invite you to use our code to help secure your own applications and to also let us know if we've missed any security vulnerabilities.

As an added bonus, we're giving away a new 15" fully loaded MacBook Pro to the first person who can breach our security. Since this guide is designed to prevent a hacker from gaining access to the full user database, we will be giving the prize to anyone that can prove access to our user database. More specifically, we aren't giving the prize for single user exploits or denial of service attacks. To win the prize, you must accomplish one of these hacks or something similar:

- Gain access to one of the servers our application is running on without detection.
- Prove an exploit that would allow retrieval of 50% or more of the user data stored in our database (via an application exploit or something similar).

To make your lives easier, we're going to give you the IP addresses of both our servers (even though in a real situation you wouldn't know the IP address of the Database Server). Those IP addresses are:

- Application Server - 192.81.133.144

- Database Server - 45.56.92.87

And finally, the front-end of our application lives here:

https://hackthis.inversoft.com

If you find an exploit or gain access to our servers, send us an email at dev@inversoft.com so we can verify it and send you the MacBook.

As you read through this guide, keep in mind that most of the instructions below are framed as if you were building an application whose architecture is similar to the one that we built. However, any server-side application that uses a database can use any or all of the techniques covered below but you might need to make some tweaks in order to get our instructions to work for your specific application.

4 Server Security

The first step to securing user data is to secure the servers you will be running your applications and databases on. The process of securing servers revolves around a couple of core guidelines:

CORE GUIDELINES

- Use the least privilege possible
- Prevent direct access when it isn't required
- Automate security checks and constraints
- Send notifications for all activity
- Identify users that can access the server

This section details a step-by-step process to secure your servers.

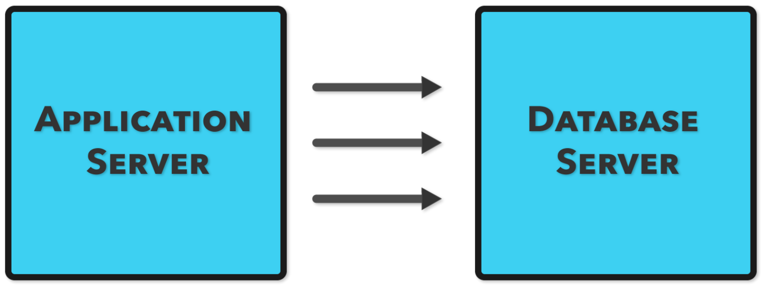
4.1 Server Architecture

In order to illustrate our thinking around server security, we need to cover what type of servers we are talking about. Since this guide is specifically about user data security, we will focus on applications that use a server side component and a database to store and manage user data. This application will allow users to register, login and perform various functions. A few examples of this type of application are:

- A web application where a user can login, access their account and perform various functions
- A mobile application that communicates with the server to manage user accounts and user data

As we mentioned above, the application we built while writing this guide is a web application that allows users to manage their “to-do list”.

Now that we know what type of servers and applications we will be securing, let’s take a look at the server architecture we used for our to-do application and the server architecture we recommend you use for your applications.



In general, you will want to separate your Database Server from your Application Server. While securing two servers instead of one does require additional work, it will give you the flexibility to control access to each server separately. This also means that if a breach does occur, the hacker will have a couple more hoops to jump through in order to gain access to your user data (we’ll discuss this later).

On the Application Server you will install the server-side code for your application. On the Database Server you will install the database that you will be using (MySQL, PostgreSQL, Mongo, Reddis, etc). Depending on your needs, you can horizontally scale either of these servers. You will want to use the exact same configuration and access controls on each server to ensure consistency. DevOps is out of scope for this guide, however there are numerous tools available to create server images and deploy multiple instances of them. Similarly, there are many tools you can use to make a configuration change to all servers without having to log into each one separately.

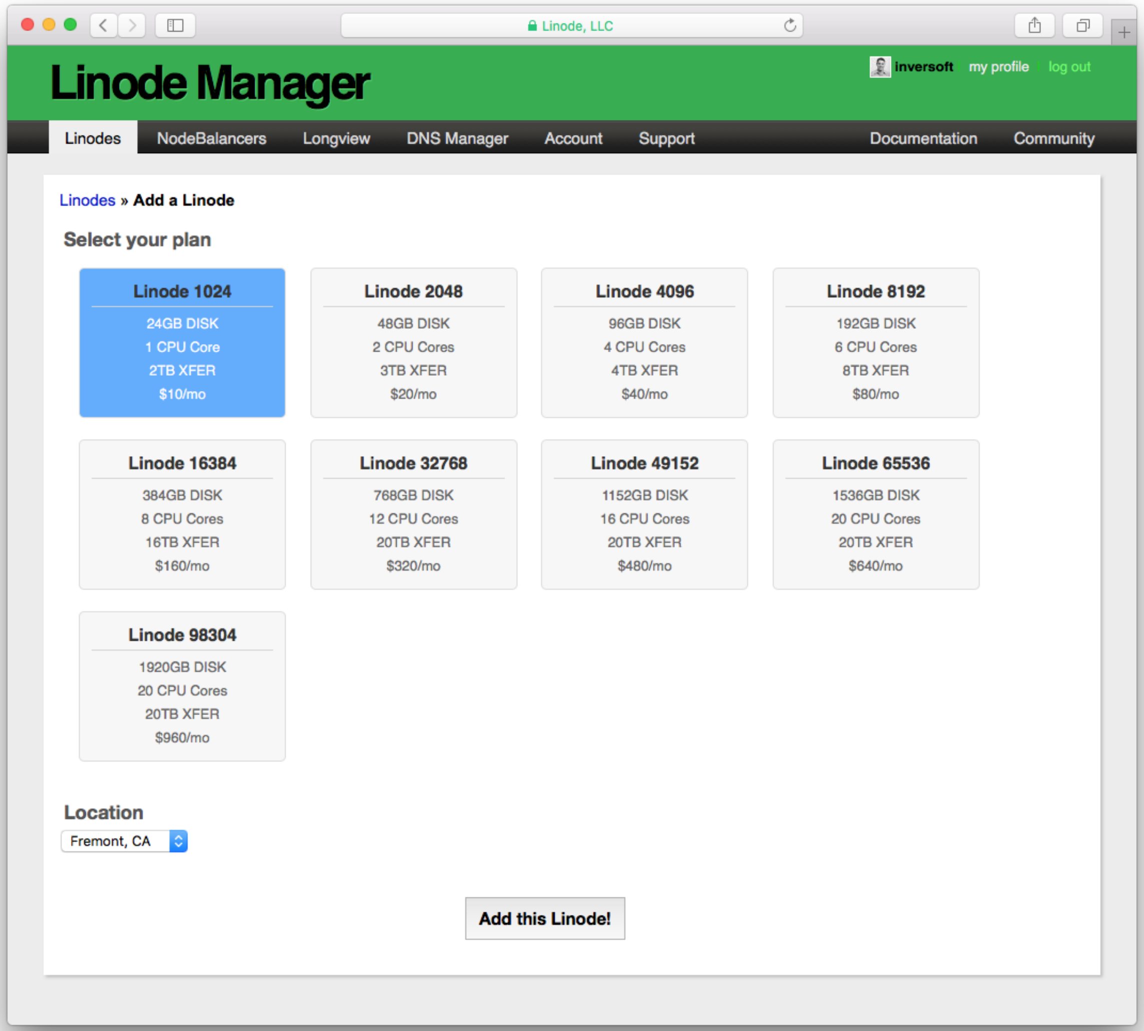
4.2 Provisioning

In this guide, we will take you through the steps to create servers using Linode, a virtual server provider. Linode provisions stock versions of most operating systems and the server is open to the world. This would be the same setup if you were running your own servers. If you use Amazon, Google or other similar providers instead, some of the security configurations below will be managed using the web based configuration systems of the hosting provider. We'll try to indicate places where providers like Amazon provide web tools to manage security.

Now that you know what the architecture looks like, let's create the two servers you need. We'll be using the Ubuntu Linux version 16.04 LTS image for this guide. Most of the file locations and instructions assume that you will be using Ubuntu. If you prefer to use CentOS or a different distribution, you will need to translate these instructions.

Our examples will use two Linode 1024 instances hosted in California. Here are the setup steps you can use to create the servers:

Step 1: Select the server type



Linodes » **Add a Linode**

Select your plan

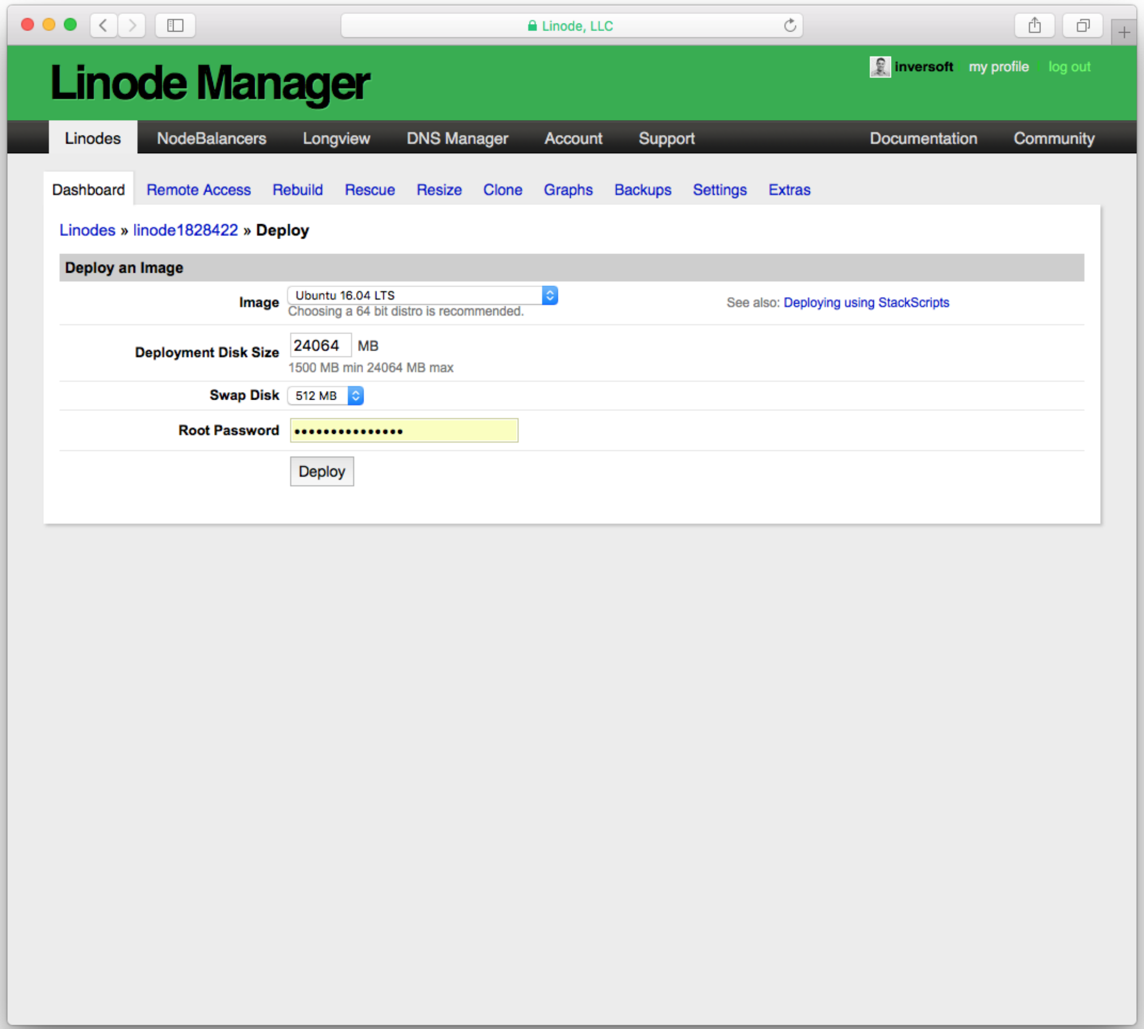
Plan	Disk	CPU	Transfer	Price
Linode 1024	24GB DISK	1 CPU Core	2TB XFER	$10/mo
Linode 2048	48GB DISK	2 CPU Cores	3TB XFER	$20/mo
Linode 4096	96GB DISK	4 CPU Cores	4TB XFER	$40/mo
Linode 8192	192GB DISK	6 CPU Cores	8TB XFER	$80/mo
Linode 16384	384GB DISK	8 CPU Cores	16TB XFER	$160/mo
Linode 32768	768GB DISK	12 CPU Cores	20TB XFER	$320/mo
Linode 49152	1152GB DISK	16 CPU Cores	20TB XFER	$480/mo
Linode 65536	1536GB DISK	20 CPU Cores	20TB XFER	$640/mo
Linode 98304	1920GB DISK	20 CPU Cores	20TB XFER	$960/mo

Location

Fremont, CA

Step 2: Deploy the Linux Operating System

In this step, be sure to select a long and secure root password. Later you will disable root passwords, but for the initial process you need to ensure the server is secure.


Linode, LLC
Linode Manager
inversoft my profile log out
Linodes NodeBalancers Longview DNS Manager Account Support Documentation Community
Dashboard Remote Access Rebuild Rescue Resize Clone Graphs Backups Settings Extras

Linodes » linode1828422 » Deploy

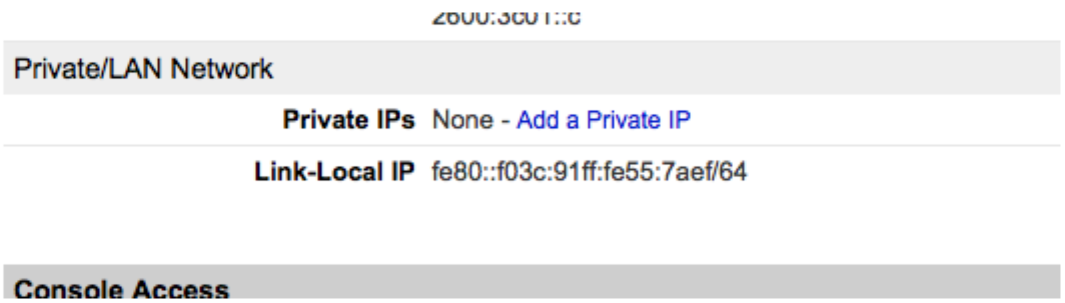
Deploy an Image

Image	Ubuntu 16.04 LTS Choosing a 64 bit distro is recommended.	See also: Deploying using StackScripts
Deployment Disk Size	24064 MB 1500 MB min 24064 MB max	
Swap Disk	512 MB	
Root Password	•••••••••••••••	
	Deploy	

Step 3: Configure a private IP address

You need to give both servers a private IP address that is not world accessible, but is accessible between Linode servers. This setting is on the Remote Access page.

LOOK A GUIDELINE!
Prevent direct access when it isn't required



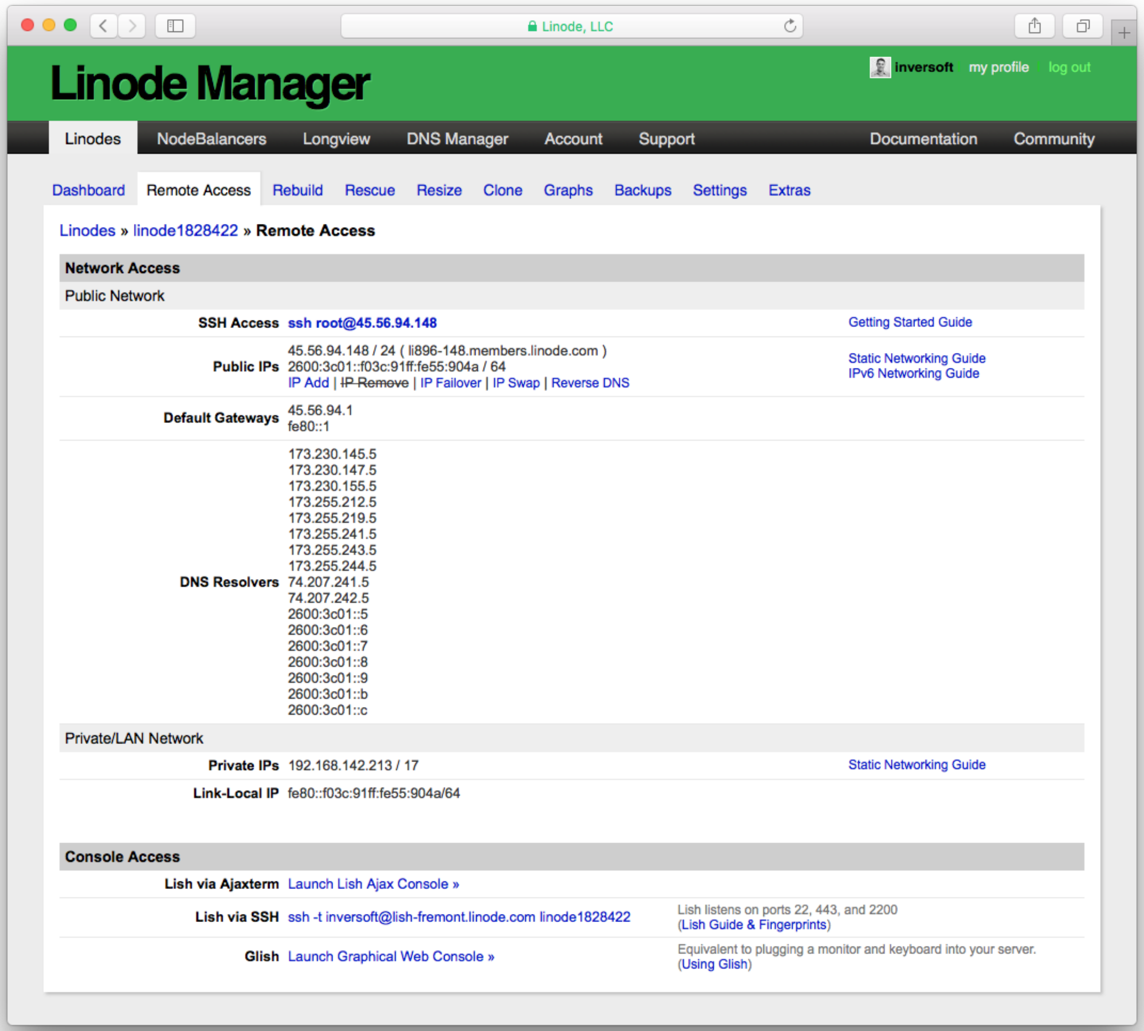
2600:3c01::c

Private/LAN Network

Private IPs	None - Add a Private IP
Link-Local IP	fe80::f03c:91ff:fe55:7aef/64

Console Access

After you add a private IP address, your configuration should look like this:



Linode, LLC
Linode Manager
inversoft my profile log out
Linodes NodeBalancers Longview DNS Manager Account Support Documentation Community
Dashboard Remote Access Rebuild Rescue Resize Clone Graphs Backups Settings Extras

Linodes » linode1828422 » **Remote Access**

Network Access

Public Network

SSH Access	ssh root@45.56.94.148	Getting Started Guide
Public IPs	45.56.94.148 / 24 (li896-148.members.linode.com) 2600:3c01::f03c:91ff:fe55:904a / 64 IP Add | ~~IP Remove~~ | IP Failover | IP Swap | Reverse DNS	Static Networking Guide IPv6 Networking Guide
Default Gateways	45.56.94.1 fe80::1	
DNS Resolvers	173.230.145.5 173.230.147.5 173.230.155.5 173.255.212.5 173.255.219.5 173.255.241.5 173.255.243.5 173.255.244.5 74.207.241.5 74.207.242.5 2600:3c01::5 2600:3c01::6 2600:3c01::7 2600:3c01::8 2600:3c01::9 2600:3c01::b 2600:3c01::c	

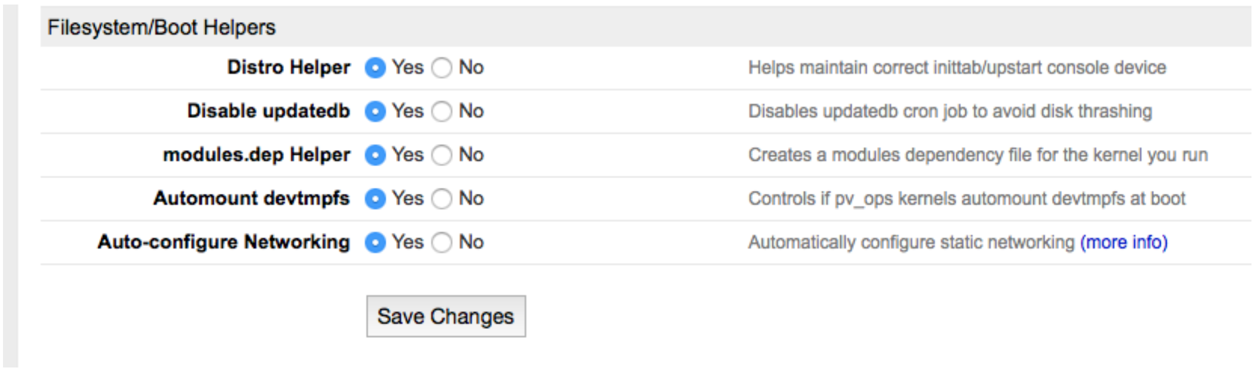
Private/LAN Network

Private IPs	192.168.142.213 / 17	Static Networking Guide
Link-Local IP	fe80::f03c:91ff:fe55:904a/64	

Console Access

Lish via Ajaxterm	Launch Lish Ajax Console »	
Lish via SSH	ssh -t inversoft@lish-fremont.linode.com linode1828422	Lish listens on ports 22, 443, and 2200 (Lish Guide & Fingerprints)
Glish	Launch Graphical Web Console »	Equivalent to plugging a monitor and keyboard into your server. (Using Glish)

Before the private IP address will take effect, you need to enable the “Auto-configure Networking” setting. From the Dashboard click the “Edit” link to the right of your configuration at the top of the page. This will take you to the configuration options. At the bottom, enable the “Auto-configure networking setting”. This option looks like this:



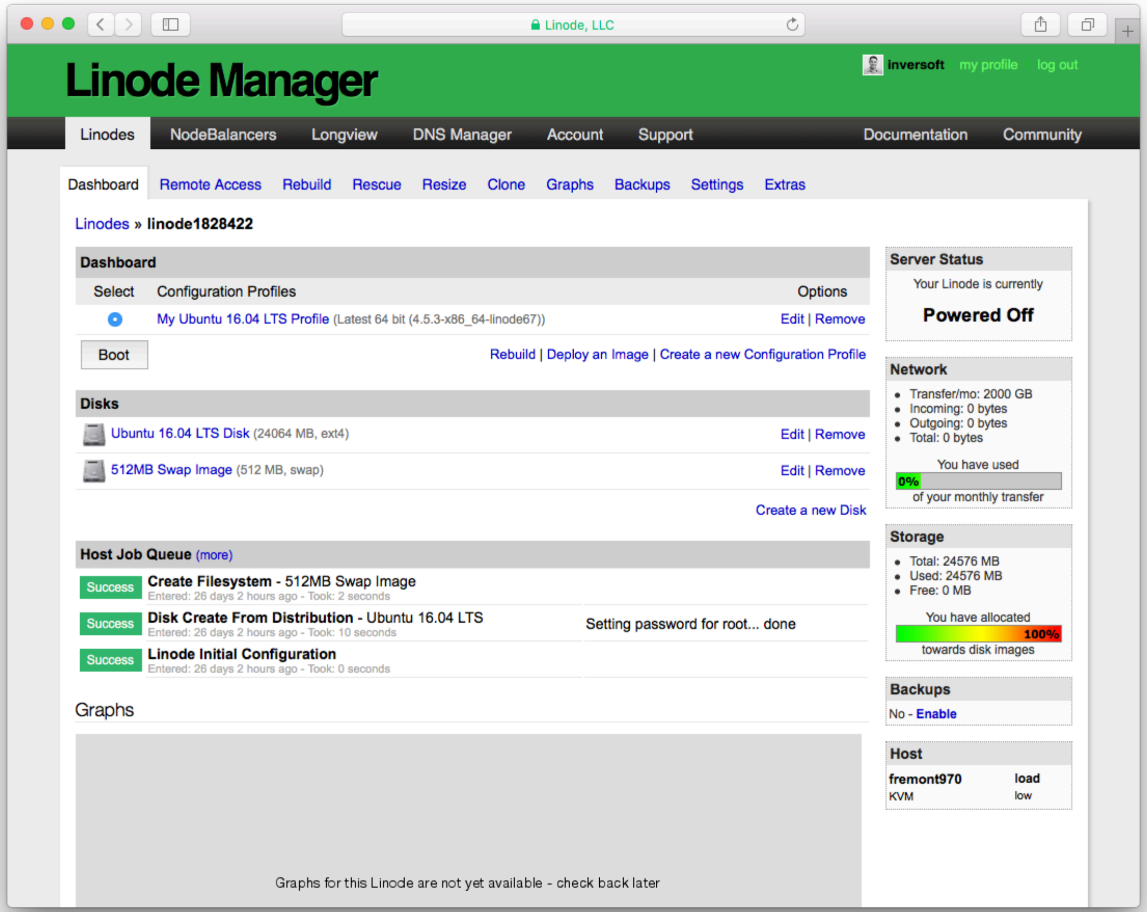
Filesystem/Boot Helpers

Distro Helper	[✓ Yes] No	Helps maintain correct inittab/upstart console device
Disable updatedb	[✓ Yes] No	Disables updatedb cron job to avoid disk thrashing
modules.dep Helper	[✓ Yes] No	Creates a modules dependency file for the kernel you run
Automount devtmpfs	[✓ Yes] No	Controls if pv_ops kernels automount devtmpfs at boot
Auto-configure Networking	[✓ Yes] No	Automatically configure static networking (more info)

Then click “Save Changes”.

Step 4: Boot the server

Now, boot your Linode server by clicking the Boot button on the details page:



Linodes » linode1828422

Dashboard

Select	Configuration Profiles	Options
●	My Ubuntu 16.04 LTS Profile (Latest 64 bit (4.5.3-x86_64-linode67))	Edit | Remove

Rebuild | Deploy an Image | Create a new Configuration Profile

Disks

Ubuntu 16.04 LTS Disk (24064 MB, ext4)	Edit | Remove
512MB Swap Image (512 MB, swap)	Edit | Remove

Create a new Disk

Host Job Queue (more)

Success	**Create Filesystem** - 512MB Swap Image Entered: 26 days 2 hours ago - Took: 2 seconds	
Success	**Disk Create From Distribution** - Ubuntu 16.04 LTS Entered: 26 days 2 hours ago - Took: 10 seconds	Setting password for root... done
Success	**Linode Initial Configuration** Entered: 26 days 2 hours ago - Took: 0 seconds	

Graphs

Graphs for this Linode are not yet available - check back later

Server Status
Your Linode is currently
Powered Off

Network
- Transfer/mo: 2000 GB
- Incoming: 0 bytes
- Outgoing: 0 bytes
- Total: 0 bytes

You have used 0% of your monthly transfer

Storage
- Total: 24576 MB
- Used: 24576 MB
- Free: 0 MB

You have allocated 100% towards disk images

Backups
No - **Enable**

Host
fremont970 KVM
load low

Both servers should now be running. The next step of the process is to lock down remote access to the servers and secure passwords and user accounts. You will need to perform all of these steps on each server to ensure they both are secure. There are numerous guides available to help you secure Linux servers, but we will cover the most common steps we use at Inversoft.

NOTE: We assume once a hacker has gained access to the server undetected, they will eventually succeed in gaining root access and will have access to everything. Therefore, we will cover some steps to secure user accounts and make the hacker's job harder, but we won't go into extreme detail here. Instead, we will focus primarily on preventing hackers from gaining access to the server in the first place.

4.3 Logging in the First Time

First, log into your server as the `root` user using the password you entered above. To log into the server, execute this command:

```
$ ssh root@192.81.133.144
```

You can leave this terminal window open while you configure the server. This makes life simpler and will allow you to fix issues if something breaks along the way.

4.4 Securing Passwords

By forcing users to select secure passwords you will prevent users with accounts on your servers from using passwords like “`password`”.

LOOK A GUIDELINE!
Automate security checks and constraints

To ensure passwords are secure, you will install a PAM module called Cracklib. This module provides the ability to set password constraints. To install this module, execute this command:

```
$ apt-get install libpam-cracklib
```

To make the most of this module, you need to edit the configuration file to require user passwords to contain complex characters including uppercase, lowercase, digits and punctuation. To set this configuration, edit the file `/etc/pam.d/common-password` like this:

```
$ nano /etc/pam.d/common-password
```

Find the line that starts with:

```
password   requisite    pam_cracklib.so
```

and replace it with:

```
password   requisite    pam_cracklib.so retry=3 minlen=10 difok=3 ucredit=-1 lcredit=-2
dcredit=-1 ocredit=-1
```

The configuration options are as follows:

- Allow 3 retries to type a new password twice before giving up (retry=3)
- Passwords must be at least 10 characters long (minlen=10)
- The new password must differ from the old password by 3 characters (difok=3)
- The new password must contain at least 1 uppercase character (ucredit=-1)
- The new password must contain at least 2 lowercase characters (lcredit=-2)
- The new password must contain at least 1 digit/number (dcredit=-1)
- The new password must contain at least 1 symbol (ocredit=-1)

You can change these values if you wish, but you should ensure users are selecting complex passwords that are hard to brute force attack.

4.5 Locking Remote Access

Remote access is always through SSH on Linux servers. In most cases, SSH is setup to allow access to user accounts via passwords. By allowing users to log in to your server using their passwords, you allow hackers to brute force attack your server from anywhere in the world. Instead, your servers should use RSA keys to allow users to login. This will only allow users to access the server from the computer that contains their private key. Since private keys will be password locked and never shared, this will ensure that only authorized users will be able to access the server.

To secure remote access to your servers, change server configuration so that SSH access requires RSA keys. If you are running in Amazon, some of this configuration is standard on their servers. You'll still want to verify your configuration using the steps below.

LOOK A GUIDELINE!
Identify users that can access the server

To start this process, you need to create a public/private key pair on your local computer. To generate a key pair on a Mac or Linux machine, execute this command:

```
$ ssh-keygen -t rsa -b 2048
```

This process will ask for a passphrase for the key. You should select a passphrase that is complex.

NOTE: A complex passphrase generally means it is 16 characters or longer and contains numbers, letters and punctuation. You should always use a complex passphrase for your keys because leaving the key without a passphrase makes it much more vulnerable overall. Even if your private key is stolen or socially hacked, a passphrase will ensure it will be much more difficult to use.

Once logged in you will create an ordinary user account so that you aren't constantly logging into the root account. This also allows you to secure the machine further. To create an ordinary user account, run this command and specify the username you want to use:

```
$ useradd -m -G sudo -s /bin/bash your-username
```

This command specifies that the user should have an extra group of sudo (the -G sudo option). This group will allow the user permissions to execute superuser commands on the server.

NOTE: Some systems have passwordless sudo enabled for the sudo group. You need to disable this so that all users are forced to type in their passwords to use the sudo command.

From your terminal window that is logged into the root account, edit the file `/etc/sudoers` like this:

```
$ nano /etc/sudoers
```

You can also use `vi` if you prefer that over `nano` for text file editing. If you aren't familiar with `vi`, `nano` will save you a lot of headache learning `vi` commands. This file should contain a line that looks like this:

```
%sudo   ALL=(ALL:ALL) ALL
```

If this line looks like this:

```
%sudo   ALL=(ALL:ALL) NOPASSWD: ALL
```

You will want to remove the `"NOPASSWD":` part of the line.

Now that the user has been created, you need to set a strong password for this user. This password should be complex. To set the password for your user, execute this command:

```
$ passwd your-username
```

In another terminal window, ensure that you can log into this user account using the password you just created:

```
$ ssh your-username@173.255.241.203
```

While you are here, you should also verify that this user has sudo access. Type this command to test for sudo access:

```
$ sudo ls /
```

You should see a directory listing of the server. However, if you see a message that says you aren't in the sudoers file, you will need to edit your `/etc/sudoers` file from the terminal that is logged in as root and add this line:

```
%sudo   ALL=(ALL:ALL) ALL
```

You can logout of this terminal by typing `exit` or hitting `ctrl-d`.

Next, copy your public key from your local computer to the server.

NOTE: Be sure to copy your public key only and NOT your private key. Here's the command to copy your public key to the server:

```
$ scp ~/.ssh/id_rsa.pub your-username@173.255.241.203:/home/your-username
```

After the public key is on the server, put the key into the `~/.ssh/authorized_keys` file so that the SSH system will use it next time you log in rather than prompting for a password. Log back into the server as the ordinary user and then execute the following commands:

```
$ mkdir .ssh
$ chmod 700 .ssh
$ mv id_rsa.pub .ssh/authorized_keys
$ chmod 600 .ssh/authorized_keys
```

Now, logout of the server and log back in as the ordinary user. You should not be prompted to type in a password. You should instead be prompted to enter the passphrase you used when creating the SSH key above. If you are prompted for a password rather than the passphrase you likely renamed your key files or specified a different name when you created them.

Therefore, you must use the -i flag to tell SSH to use the proper identity file for this server like this:

```
$ ssh -i server_rsa your-username@173.255.241.203
```

The server is now ready to have the SSH system secured. In the terminal that is still logged in as root, edit the `/etc/ssh/sshd_config` file. You need to change a few lines in this file to prevent anyone from logging into the server without a SSH key pair and to prevent the root user from logging into the server via SSH. Find these configuration options and change their values to those below. Some of these options might be commented out because they are prefixed with a # character. Simply remove the # character and change the setting.

```
PermitRootLogin no
PasswordAuthentication no
```

Save this file and restart the SSH service like this:

```
$ service ssh restart
```

You can verify that your server is now secure by attempting to SSH to it as the root user. You should receive an error message like this:

```
Permission denied (publickey).
```

4.6 SSH Agents

You might be wondering what will happen if you SSH to the Application Server and then try to SSH from there to the Database Server. By default, you won't be able to do this because your private key won't be on the Application Server. However, SSH has a feature that allows you to accomplish this without copying your private key all over the place.

The feature you will use is called SSH agent. By enabling an SSH agent, you will be able to log into any server that has your public key setup in the `authorized_keys` file (as long as you start from a computer that has your private key).

To setup an SSH agent, add your private key to the agent by running this command:

```
$ ssh-add
```

This will add your default private key to the SSH agent.

NOTE: If you are on a Mac, you don't need to run this command. OSX will automatically prompt for your private key passphrase and add this key to your SSH agent.

You need to enable your SSH agent when you SSH to a server. To do this, you will use the `-A` option like this:

```
$ ssh -A -i server_rsa your-username@173.255.241.203
```

You can also add an alias for the SSH command to automatically add the `-A` option in your shell's configuration file.

4.7 Locking the Root Account

Disable the root user's login to prevent users from trying to brute force attack the root user if they ever gain access to the server. You might be concerned that if you lock the root account you won't be able to administer the server easily. Even if the root user account is locked, you'll still be able to use the root account via the command "`sudo su -`", but you won't be able to log in directly to the root user's account. To lock the root user's login, type this command into the root user terminal:

LOOK A GUIDELINE!
Prevent direct access when it isn't required

```
$ usermod -p '*' root
```

4.8 Two-Factor Authentication

Since we are going for über security, let's get crazy and add one more layer of authentication to your remote access. Add two-factor authentication to the server; this will require users to type in a 6-digit code from their mobile phone in order to log in. Even if a user's private key is compromised, two-factor authentication ensures a hacker would still need the user's mobile phone to log

LOOK A GUIDELINE!
Identify users that can access the server

into the server.

First, you will need to install a two-factor app on your smartphone. There are many two-factor applications available for iOS and Android. Next, install the Google Authenticator two-factor PAM module on the server by executing this command in the root user terminal:

```
$ apt-get install libpam-google-authenticator
```

Edit the PAM configuration file for SSH like this:

```
$ nano /etc/pam.d/sshd
```

Add this line to the top of the file:

```
auth [success=done new_authtok_reqd=done default=die] pam_google_authenticator.so nullok
```

Save this file and then open the SSH configuration file like this:

```
$ nano /etc/ssh/sshd_config
```

Find and modify or add the configuration parameters below:

```
ChallengeResponseAuthentication yes
AuthenticationMethods publickey,keyboard-interactive
```

Before enabling this module for a user, install the network time system so that the server's clock is updated and consistent with the atomic clock. Since the two-factor authentication system uses a time-based algorithm, this will ensure that the server clock remains in sync with your smartphone. To install this package, type this command:

```
$ apt-get install ntp
```

Now restart the SSH service to pick up these changes like this:

```
$ service ssh restart
```

Finally, log into the ordinary user account and execute this command to enable two-factor authentication:

```
$ google-authenticator -l 'your-username@Application Server'
```

This will display a QRCode that you can scan from the two-factor authentication app on your smartphone. It will also ask you additional questions to determine how you want to handle time-skew and brute force login attempts. Additionally, there will be 5 emergency scratch codes generated for your account. You should write these down in a secure location such as iCloud Secure Notes.

Before logging out of this terminal, open a new terminal window and verify that you can still login to the ordinary user account with your SSH key and the two-factor code from your smartphone. If you ever have problems logging in, you can use one of the scratch codes that you saved off above. This will grant you access to your account and allow you to fix the two-factor authentication issue.

If you install the two-factor authentication as described above and a user doesn't enable it for their account, everything will still work fine. They will be able to log in using just their SSH keys.

4.9 Firewalling

This section covers how to prevent access to the servers except on specific ports and from specific IP addresses.

LOOK A GUIDELINE!
Prevent direct access when it isn't required

Configuration are different for each server to ensure that each server prevents as much direct access as possible. For the Application Server, you will lock down all ports except those needed to access your application's server-side components and to access the server via SSH. For the Database Server, you will lock down all ports except the database port and SSH port. In addition, on the Database Server you will also prevent access from everywhere except the Application Server. If you are using Amazon, they provide a web based firewall configuration for servers and you will use that rather than IPTables to manage access to your servers.

First, install the firewall package. This package will ensure that the firewall rules are loaded each time the server is restarted. To install this package, execute this command in the root terminal:

```
$ apt-get install iptables-persistent
```

Make sure you save both the rules.v4 and rules.v6 file during this installation process.

Now, setup the firewall on the Application Server. Your application isn't running yet, but when it does run it will be listening on two ports. These will vary based on your setup, but our examples will use 3003 for HTTPS web requests and port 3000 for HTTP web requests. We will actually forward requests from the standard ports 80 for HTTP and 443 for HTTPS to ports 3000 and 3003 respectively. This forwarding is covered later.

Likewise, the Application Server will listen on port 22 for SSH requests. Since you installed the persistent IPTables package, you can simply edit the rules file to set your rules. Copy and paste the following content into the `/etc/iptables/rules.v4` file (via `nano` or any other editor):

```
*filter
# Allow all outgoing, but drop incoming and forwarding packets by default
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

# Custom per-protocol chains
:UDP - [0:0]
:TCP - [0:0]
:ICMP - [0:0]

# Acceptable UDP traffic

# Acceptable TCP traffic
-A TCP -p tcp --dport 22 -j ACCEPT
-A TCP -p tcp --dport 443 -j ACCEPT
-A TCP -p tcp --dport 80 -j ACCEPT
-A TCP -p tcp --dport 3000 -j ACCEPT
-A TCP -p tcp --dport 3003 -j ACCEPT

# Acceptable ICMP traffic

# Boilerplate acceptance policy
-A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
-A INPUT -i lo -j ACCEPT

# Drop invalid packets
-A INPUT -m conntrack --ctstate INVALID -j DROP

# Pass traffic to protocol-specific chains
## Only allow new connections (established and related should already be handled)
## For TCP, additionally only allow new SYN packets since that is the only valid
## method for establishing a new TCP connection
-A INPUT -p udp -m conntrack --ctstate NEW -j UDP
-A INPUT -p tcp --syn -m conntrack --ctstate NEW -j TCP
-A INPUT -p icmp -m conntrack --ctstate NEW -j ICMP

# Commit the changes
COMMIT
```

```
*raw
:PREROUTING ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
COMMIT

*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

-A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 3000
-A PREROUTING -p tcp --dport 443 -j REDIRECT --to-port 3003

# Commit the changes
COMMIT

*security
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
COMMIT

*mangle
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
COMMIT
```

A complete explanation of the IPTables rule file format is out of scope for this guide however we will cover some of the basics so that you can configure your file to suit your needs. IPTables rule files are broken down into Tables. Tables are defined using an asterisk character followed by the table name. Everything after the table definition is included in that table's rules until the `COMMIT` command is hit.

The filter table is the most important table above, but the nat table is also used and we'll cover that shortly. A filter table is defined like this:

```
*filter
…
COMMIT
```

We'll ignore the raw, security and mangle tables for now. You can find more information about them online if you are interested.

A table can hold one or more chains. There are a number of predefined chains in each table, but for the filter table the three chains are `INPUT`, `OUTPUT` and `FORWARD`. In the nat table, we will use the `PREROUTING` chain as well. Here are the definitions of these chains:

- `INPUT` - this chain is used when packets are sent from an external source to the server
- `OUTPUT` - this chain is used when packets are sent from the server to an external source
- `FORWARD` - this chain is used when the server is forwarding packets between two external computers (rarely used)
- `PREROUTING` - this chain is used to modify incoming packets before they are delivered

Chains are defined using a colon character followed by the name of the chain. Here's an example of defining a new chain called `FOOBAR`:

```
:FOOBAR [0:0]
```

NOTE: The numbers inside the brackets are the number of bytes sent and received on the chain. They are mostly for informational purposes and in the example above, we initialize everything to zero.

It is also important to keep in mind that a connection based protocol like TCP will use both the `INPUT` and the `OUTPUT` chain because packets flow in both directions.

When network packets arrive at the server, they are handled by one of these three chains. You can define all of your rules directly on these chains or you can define new chains and JUMP to them from the three predefined chains. The rules file above defines three new chains, one for each of the protocols you are probably interested in handling on your servers. The rules file also defines how the predefined chains `JUMP` to the new chains. The chains are defined at the top of the file and are named `TCP`, `UDP` and `ICMP` to map to each of the protocols.

`The JUMP` rules are midway through the file and look like this:

```
-A INPUT -p udp -m conntrack --ctstate NEW -j UDP
```

Here's how this rule reads: "Append to the `INPUT` chain (`-A INPUT`), for protocol UDP (`-p udp`), when new connections are made (`-m conntrack -cstate NEW`), jump to the `UDP` chain (`-j UDP`). As you can see, there is a similar rule for each of the new chains we created for each protocol.

Each rule must define a policy that determines how packets are handled. This is done by indicating one of these 3 policies:

- `ACCEPT` - the packet is allowed and whatever service is listening for network traffic on the specific port will be sent the packets
- `DROP` - the packet is completely ignored; the external computer that sent the packet will not receive a response and will be forced to time-out the connection
- `REJECT` - a response is sent to the external computer indicating that the server has rejected the packet

In the configuration above, we initialize the `INPUT` and `FORWARD` chains to `DROP` everything by default. This prevents your server from accepting packets unless you explicitly define a rule. The `OUTPUT` chain on the other hand is set to `ACCEPT` by default. In most cases, if you are on the server, you'll likely want to access other servers. You might need to do a DNS lookup or download packages to install, but it is generally safe to leave the `OUTPUT` chain open.

The file above sets up a few default rules for the predefined `INPUT` chain. These rules are:

```
# Boilerplate acceptance policy
-A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
-A INPUT -i lo -j ACCEPT

# Drop invalid packets
-A INPUT -m conntrack --ctstate INVALID -j DROP
```

Here's a quick explanation of these rules:

- All traffic on the loopback interface (packets going from the server to itself) are allowed.
- All traffic on connections that have already been established or related connections are allowed.
- All invalid packets are dropped. Switches and other networking systems can mark packets as invalid, so ignoring those packets won't cause any issues.

The final rules are setup for each of the protocols you are interested in. By default, UDP and ICMP traffic are completely ignored. The file above assumes that you will only be using TCP connections to the server, so it only defines rules for that filter chain. If you need to use UDP or ICMP, you can add additional rules for those protocols. Here are the rules for the TCP chain:

```
-A TCP -p tcp --dport 22 -j ACCEPT
-A TCP -p tcp --dport 443 -j ACCEPT
-A TCP -p tcp --dport 80 -j ACCEPT
-A TCP -p tcp --dport 3000 -j ACCEPT
-A TCP -p tcp --dport 3003 -j ACCEPT
```

These rules read like this, "for TCP packets (`-A TCP -p tcp`), on port 22 (`--dport 22`), accept them (`-j ACCEPT`)". There is a separate rule for each port that should be open on the server. You'll notice that the server listens on the standard HTTP and HTTPS ports, but also port 3000 and 3003, which are the ports our example application will be listening on.

In addition to the filter table, the configuration above also makes use of the nat table. The nat table is used because later in this guide, you will run your application as a non-privileged user (i.e. not the root user). This is to ensure that breaches to the application won't allow hackers full access to the server. Linux servers don't allow non-privileged users to bind ports below 1000. Therefore, the IPTables configuration forwards traffic on port 80 to port 3000 and traffic on port 443 to port 3003. To accomplish this port forwarding, we make use of the `PREROUTING` chain in the nat table.

Under the nat table, there are two rules defined:

```
-A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 3000
-A PREROUTING -p tcp --dport 443 -j REDIRECT --to-port 3003
```

These rules read like this, "append to the PREROUTING chain (`-A PREROUTING`) for TCP packets (`-p tcp`), forward destination port 80 (`--dport 80 -j REDIRECT`) to port 3000 (`--to-port 3000`).

Now that the rules are setup, load this configuration by executing this command:

```
$ service netfilter-persistent reload
```

Next, configure the firewall on the Database Server. Each database will have different ports that it needs open. You will only allow connections from your Application Server's private IP address, which you will need to know and insert into the rules below. Here are the IPTable rules for the Database Server.

```
*filter
# Allow all outgoing, but drop incoming and forwarding packets by default
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

# Custom per-protocol chains
:UDP - [0:0]
:TCP - [0:0]
:ICMP - [0:0]

# Acceptable UDP traffic
# Acceptable TCP traffic
-A TCP -p tcp --dport 22 -s 192.168.1.12 -j ACCEPT
-A TCP -p tcp --dport 3306 -s 192.168.1.12 -j ACCEPT

# Acceptable ICMP traffic

# Boilerplate acceptance policy
-A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
-A INPUT -i lo -j ACCEPT

# Drop invalid packets
-A INPUT -m conntrack --ctstate INVALID -j DROP

# Pass traffic to protocol-specific chains
## Only allow new connections (established and related should already be handled)
## For TCP, additionally only allow new SYN packets since that is the only valid
## method for establishing a new TCP connection
-A INPUT -p udp -m conntrack --ctstate NEW -j UDP
-A INPUT -p tcp --syn -m conntrack --ctstate NEW -j TCP
-A INPUT -p icmp -m conntrack --ctstate NEW -j ICMP

# Commit the changes
COMMIT

*raw
:PREROUTING ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
COMMIT

*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
COMMIT

*security
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
COMMIT

*mangle
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
COMMIT
```

The primary difference between the Application Server's and the Database Server's IPTables is that the Database Server only allows connections on port 22 for SSH and 3306 for MySQL. Plus, only connections that originate from the Application Server to the Database Server are allowed. The lines that setup this configuration are:

```
-A TCP -p tcp --dport 22 -s 192.168.1.12 -j ACCEPT
-A TCP -p tcp --dport 3306 -s 192.168.1.12 -j ACCEPT
```

You can also define multiple IP addresses using a comma separated list or subnet masks (e.g. 192.168.1.2,192.168.1.3 or 192.168.1.0/24).

It might seem like a pain to have to log into the Application Server in order to log into the Database Server. Nevertheless, this is usually a good idea. By doing so, the Database Server (where all your user data is stored) is protected from brute force attacks. If you put yourself in a hacker's shoes this security measure makes sense:

> *You have finally managed to log into the Application Server because the Database Server wasn't publicly accessible. However, you still have the daunting task of hacking either the MySQL password on the current server by gaining root access or hacking the login to the Database Server.*

Both are challenging feats and it is likely the hacker will be discovered well before they accomplish either task.

4.10 Intrusion Detection

Since most servers aren't actively logged into on a frequent basis, you should be notified when anyone logs into the server. There are many ways to handle this, but we will cover how to setup notifications using a tool called Monit. Monit allows you to watch log files for messages that contain certain text and generate alerts. To install Monit, execute this command in the root terminal window:

LOOK A GUIDELINE!
Send notifications for all activity

```
$ apt-get install monit
```

After Monit is installed, you can create a configuration file to monitor SSH login. Copy and paste the configuration below into the file `/etc/monit/conf.d/ssh-logins`:

```
check file ssh_logins with path /var/log/auth.log
```

```
    # Whitelist IPs here if you want
    # ignore match "/etc/monit/whitelist_ips.regex"
    if match "Accepted keyboard-interactive/pam" then alert
```

This file states that Monit should watch the file `/var/log/auth.log` and if it sees new lines that contain the expression "`Accepted keyboard-interactive/pam`" an alert should be generated. Since you configured two-factor authentication above, the SSH system will insert a log into this file that matches this pattern. If you did not enable two-factor authentication above (and are only using key based authentication) your configuration should instead match on the string "`Accepted publickey`".

You can also whitelist certain IP addresses by uncommenting the ignore match line and creating the file `/etc/monit/whitelist_ips.regex`. This file would contain regular expressions for each whitelisted IP on a separate line.

The last step to configuring Monit is to edit the file `/etc/monit/monitrc`. Add the lines below to configure Monit to send email alerts to you. This configuration will vary depending on your SMTP mail server, the authentication required for your SMTP server and your email address. Our examples use Sendgrid for sending emails and my email address.

```
set mailserver smtp.sendgrid.net port 587 username "<sendgrid-username>" password
"<sendgrid-password>" using tlsv12
set alert brian@inversoft.com not on { instance, action }
```

The second line tells Monit to email me whenever alerts happen, but to ignore alerts I created manually. This will reduce spamming your inbox if you choose to use Monit for other tasks such as process watching and restarting.

You might want to receive Monit notifications quickly. By default, Monit runs checks every 2 minutes. Rather than wait two minutes, you can change this setting to something smaller like 5 seconds. This configuration parameter is near the top of the `/etc/monit/monitrc` file and looks like this:

```
set daemon 5
```

Finally, restart the Monit daemon so that it picks up the configuration change:

```
$ service monit restart
```

As an added option, you can have Monit send push notifications to your mobile devices and/or send notifications to a Slack channel. One option for handling push notifications is using the Pushover service. To implement this you need to change the last line in `/etc/monit/conf.d/ssh-logins` to this:

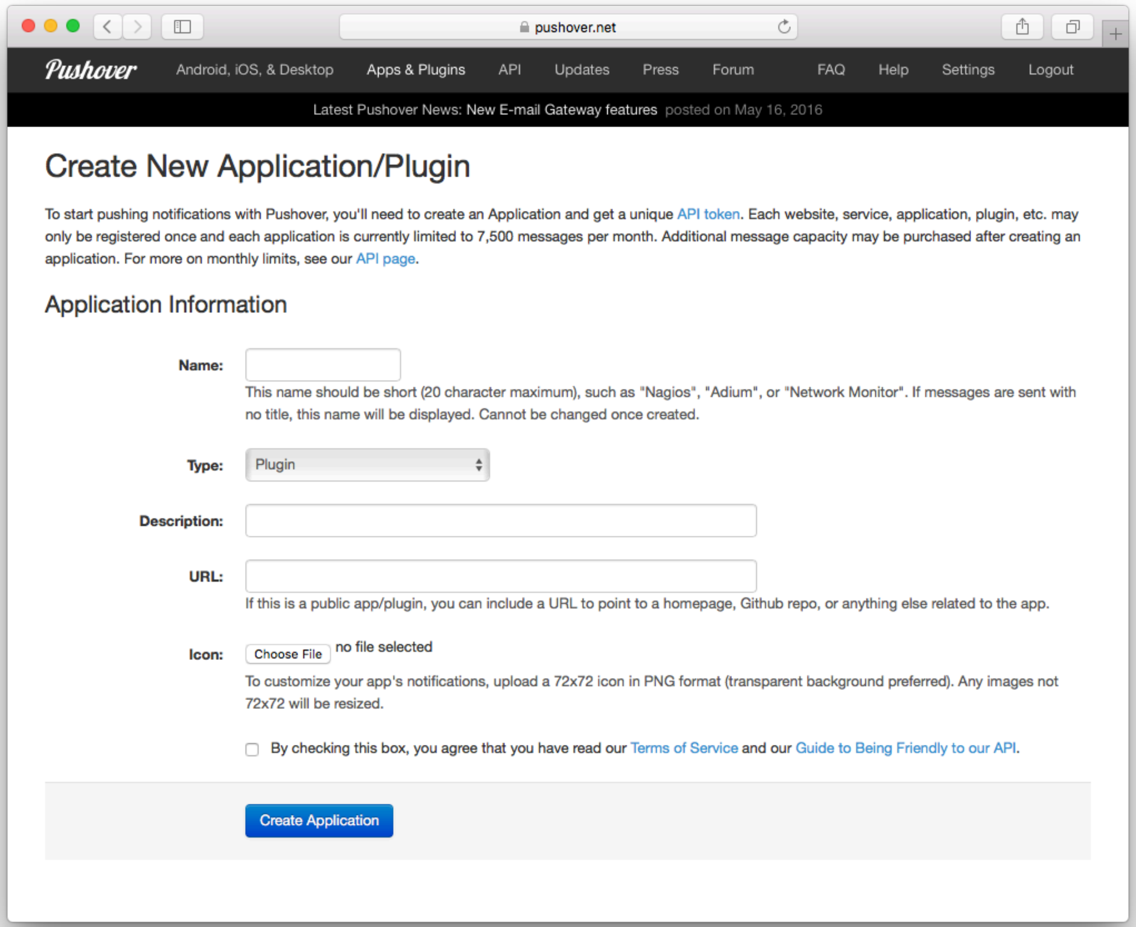
```
if match "Accepted keyboard-interactive/pam" then exec "/etc/monit/monit-slack-pushover.rb"
```

Then copy this Ruby script into the file `/etc/monit/monit-slack-pushover.rb`:

```
#!/usr/bin/ruby

require 'net/https'
require 'json'

slack_webhook_url=""
slack_enabled=true
pushover_application=""
pushover_user=""
pushover_enabled=true

def log(message)
  open('/var/log/monit.log', 'a') { |f|
    f.puts message
  }
end

if slack_enabled
  begin
    uri = URI.parse(slack_webhook_url)
    Net::HTTP.start(uri.host, uri.port, {use_ssl: true}) { |http|
      request = Net::HTTP::Post.new(uri.request_uri, {'Content-Type' => 'application/json'})
      request.body = {
          :text => "[#{ENV['MONIT_HOST']}] #{ENV['MONIT_SERVICE']} -
#{ENV['MONIT_DESCRIPTION']}"
      }.to_json
      response = http.request(request)
      log("Response from Slack [#{response.code}] [#{response.body}]")
    }
  rescue Exception => e
    log("Exception while calling Slack [#{e.message}]")
  end
end

if pushover_enabled
  begin
    uri = URI.parse("https://api.pushover.net/1/messages.json")
    Net::HTTP.start(uri.host, uri.port, {use_ssl: true}) { |http|
      request = Net::HTTP::Post.new(uri.request_uri, {'Content-Type' => 'multipart/form-
data'})
      request.set_form_data(token: pushover_application, user: pushover_user, message:
```

```
"[#{ENV['MONIT_HOST']}] #{ENV['MONIT_SERVICE']} - #{ENV['MONIT_DESCRIPTION']}")
      response = http.request(request)
      log("Response from Pushover [#{response.code}] [#{response.body}]")
    }
  rescue Exception => e
    log("Exception while calling Pushover [#{e.message}]")
  end
end
```

You must configure the Pushover and Slack variables defined at the top of the file for the integrations to work properly. For the Pushover integration, create a Pushover account and a Pushover application. The Pushover application creation looks like this:



Latest Pushover News: New E-mail Gateway features posted on May 16, 2016

Create New Application/Plugin

To start pushing notifications with Pushover, you'll need to create an Application and get a unique API token. Each website, service, application, plugin, etc. may only be registered once and each application is currently limited to 7,500 messages per month. Additional message capacity may be purchased after creating an application. For more on monthly limits, see our API page.

Application Information

Name:
This name should be short (20 character maximum), such as "Nagios", "Adium", or "Network Monitor". If messages are sent with no title, this name will be displayed. Cannot be changed once created.

Type: Plugin

Description:

URL:
If this is a public app/plugin, you can include a URL to point to a homepage, Github repo, or anything else related to the app.

Icon: Choose File no file selected
To customize your app's notifications, upload a 72x72 icon in PNG format (transparent background preferred). Any images not 72x72 will be resized.

☐ By checking this box, you agree that you have read our Terms of Service and our Guide to Being Friendly to our API.

Create Application

Give your Pushover application a name, description and URL. Select “Script” from the Type select box.

Once you create the Pushover application it will have an id associated with it. Take the Pushover application and user ids and put them into the variables `pushover_application` and `pushover_user` in the script above. Enable Pushover in the script by setting the variable `pushover_enabled` to true.

For Slack integration, configure an Incoming Webhook using these steps:

1. Open your Slack interface and click on the name in the top left corner.
2. Select “Apps & Integrations”.
3. Click “Build your own” in top right corner.
4. Click “Make a Customer Integration” under “Just for my team”.
    - Here you’ll select the channel you want to post to. You’ll have some other options to set if you want. All you need from here is the integration URL.
5. Copy and paste that URL into the variable `slack_webhook_url` in the script above.

To ensure the Ruby script above will run properly, make sure you have Ruby installed and the script is executable by executing these commands:

```
$ apt-get install ruby
$ chmod +x /etc/monit/monit-slack-pushover.rb
```

Finally, restart Monit again using this command:

```
$ service monit restart
```

4.11 Database Server Specific Security

This section will cover security measures that are specific to the Database Server using MySQL. You can translate these instructions for nearly any database solution you use.

4.11.1 MySQL

First, install MySQL. To install MySQL on the Database Server, execute this command in the root terminal window:

```
$ apt-get install mysql-server
```

During this installation process you will be prompted to create a password for the MySQL root user. Make sure that you select a very strong password and store it somewhere safe.

By default, MySQL only listens on the loopback interface. Change this configuration so that it will listen on the private IP address you created for the Application Server above. Open the file `/etc/mysql/mysql.conf.d/mysqld.cnf` (**NOTE:** This is a Ubuntu/Debian specific location and your server might have this file in another directory with a different name) and find and edit the listen configuration setting to this value:

```
bind-address = 192.168.142.213
```

This is the private IP address from our example above. You will need to change this to your private IP address. This will ensure that only servers on the same LAN will be able to connect to the database.

LOOK A GUIDELINE!

Prevent direct access when it isn't required

Lastly, verify that the MySQL data directory is properly secured and the MySQL user cannot be logged into.

This is the default on nearly all Linux distributions, but let's double check. In the MySQL configuration file, you can determine where the data directory is by finding the configuration property datadir. On most Debian distributions, the data directory is located at `/var/lib/mysql`. Do a long directory listing in `/var/lib` to see if the directory is secure. Here's the command and what the output should look like:

```
$ ls -alu /var/lib
…
drwx------  5 mysql mysql 4096 Apr 28 19:32 mysql
drwx------  2 mysql mysql 4096 Apr 28 19:33 mysql-files
drwx------  2 mysql mysql 4096 Apr 28 19:33 mysql-keyring
```

Your directories should look similar. They should all be owned by the user and group named mysql. The permissions should be read, write, execute for the user only (that's what "`drwx------`" means). If this is not how your directories are set, you can change the ownership and permissions for these directories by executing these commands:

```
$ chown -R mysql:mysql /var/lib/mysql*
$ chmod -R go-rwx /var/lib/mysql*
```

Also, open the file `/etc/password` and ensure that the line that starts with "`mysql`" ends with "`/bin/false`". It should look something like this:

```
mysql:x:110:117:MySQL Server,,,:/nonexistent:/bin/false
```

If the MySQL user's line doesn't end in "`/bin/false`", you can edit this file manually and change it using nano or vi.

4.11.2 Backups

You'll want to make backups of your Database Server and store them offsite in case of a server failure. One issue that arises with backups is that they are often not secured. The server they are stored on might have been built by a hosting provider or some other third party therefore, the Backup Server might not have the level of security you require. You should encrypt database backups before transferring them offsite. This reduces the likelihood that a Backup Server breach will allow the hacker to gain access to your data.

LOOK A GUIDELINE!
Automate security checks and constraints

First, set up encryption keys. To properly encrypt backups from a script, you need a public/private key pair. As the root user on your Database Server, change to the root user's home directory and then execute these commands:

```
$ openssl genrsa -out private-key.pem 2048
$ openssl rsa -in private-key.pem -out public-key.pem -outform PEM -pubout
```

Now that you have the key pair, leave the `public-key.pem` on the Database Server and move the `private-key.pem` file to the Backup Server (or any other secure location). Now you are ready to backup the database and encrypt the backup file.

First, get the backups running on your Database Server. The simplest way to run a backup is using the mysqldump command via cron. To start, you need to create a shell script in a protected directory. For this example, we are going to use the root user's home directory. From the root user terminal, type these commands:

```
$ cd
$ mkdir bin
$ touch bin/backup.sh
$ chmod +x bin/backup.sh
$ nano bin/backup.sh
```

In this file, you will need to dump the contents of your database to a file. The command to do this is mysqldump. Here's an example script that creates a backup of a database called `production` to the `/tmp/backups` directory:

```
#!/bin/bash

BACKUP_DIR=/tmp/backups
DATE=`date +%Y%m%d`
BACKUP_FILE="${DATE}.tar.gz"
BACKUP_FILE_ENCRYPTED="${DATE}.tar.gz.enc"
BACKUP_PASSPHRASE_FILE="${DATE}.passphrase"
BACKUP_PASSPHRASE_FILE_ENCRYPTED="${DATE}.passphrase.enc"

# Make the directory just in case it doesn't exist
mkdir ${BACKUP_DIR}
cd ${BACKUP_DIR}

# Delete the oldest files by only listing out everything older than the newest 7 files
ls *.gz.enc | sort | tail -n +7 | xargs rm
ls *.gz.passphrase.enc | sort | tail -n +7 | xargs rm

# Backup the MySQL databases
mysqldump -u<username> -p<password> production > production.sql

# Tar GZ everything (modify this line to include more files and directories in the backup)
tar -pczf ${BACKUP_FILE} *.sql

# Generate a random passphrase
openssl rand 32 -out ${BACKUP_PASSPHRASE_FILE}

# Encrypt the backup tar.gz
openssl enc -aes-256-cbc -pass file:${BACKUP_PASSPHRASE_FILE} < ${BACKUP_FILE} >
${BACKUP_FILE_ENCRYPTED}

# Encrypt the passphrase
openssl rsautl -encrypt -pubin -inkey ~/public-key.pem < ${BACKUP_PASSPHRASE_FILE} >
${BACKUP_PASSPHRASE_FILE_ENCRYPTED}

# Clean up
rm ${BACKUP_FILE} ${BACKUP_PASSPHRASE_FILE}

# Copy offsite
#scp ${BACKUP_FILE_ENCRYPTED} ${BACKUP_PASSPHRASE_FILE_ENCRYPTED} <username>@<backup-
server>:backups/.
```

You might want to backup additional files and directories in this script as well. The last line is used to copy the file offsite to a Backup Server. If you choose to store your backups offsite, you will need to create a Backup Server and secure it using a similar process as above. The setup of the Backup Server is out of scope for this guide, but the main differences in the process above are as follows:

- The Backup Server will not use two-factor authentication.
- The Backup Server will only allow SSH access from the Database Server and all other ports will be locked down.
- The Database Server will be setup with a SSH public/private key pair specifically for the Backup Server. This key pair will be stored in the root user's `~/.ssh` folder on the Database Server and the public key will be stored on the Backup Server in a special user account just for backups.
- The SSH key cannot have a passphrase since that would prevent the backup script from running automatically.

The last step to setup your backups is to enable them via cron. To accomplish this, type this command in as the root user on the Database Server:

```
$ crontab -e
```

You can select the editor you want to use and then add this line to the end of the crontab file:

```
1 0 * * * /root/bin/backup.sh
```

Save this file. Your backups will run each night at 12:01am.

If you ever need to use a backup, you will need to decrypt it first. To decrypt a backup, you will need the private key you created above and the two encrypted files that the backup script created (the encrypted tar.gz and the encrypted passphrase file). If you are in a directory that contains all of these files, execute these commands to decrypt the backup (you'll need to change the file names below to match your file names):

```
$ openssl rsautl -decrypt -inkey private-key.pem < 20160502.passphrase.enc >
20160502.passphrase
$ openssl enc -aes-256-cbc -d -pass file:20160502.passphrase < 20160502.tar.gz.enc >
20160502.tar.gz
```

5 Application Security

Now that your servers are nicely secured, let's take a look at securing the application. When we discuss application level security, we are talking about hacks and exploits that would allow hackers access to all or most of the user data. We won't be focusing on attacks that might allow a hacker access to a single user account.

Application Security centers around a set of guidelines:

CORE GUIDELINES

- Use the least privilege possible
- Encrypt all communications
- Automate security checks and constraints
- Cleanse user input
- Keep it on the server

This section will cover the different aspects of securing an application.

5.1 Application User

When you install and start your application on the Application Server you have a couple of options. You can run the application using root, the ordinary user you created above or a new user. We recommend that you create a new user to run your application for a couple of reasons:

LOOK A GUIDELINE!
Use the least privilege possible

- Creating a new user prevents the application from having root privileges to the server.
- If the application has root privileges on the server and it has a security flaw, hackers might exploit that flaw to gain access to the server.
- If you run the application as the ordinary user you created above and a hacker gains access to that account, they will have access to the application configuration. This usually means they will have access to the user database.

You should instead create a new user that is a locked account just for your application. To create this user, execute this command as root on the Application Server (you can change the username if you want):

```
$ useradd -M -s /bin/false application
```

This command creates a user named `application` that has no password and no home directory. It cannot be logged into. However, you will be able to start your application as this user using the `sudo` or `su` commands like this:

```
#!/bin/bash

sudo -u application nodejs /usr/local/application/app.js
```

5.2 SSL

The first level of defense for any application is to use SSL (we use the term SSL to mean both SSL and TLS) for all network traffic to the web server. Without SSL encryption, a hacker could intercept network traffic in and out of the server and store it. Since this network traffic would contain usernames, passwords and other user data this would generally be the same level of breach as a hacker gaining access to the user database.

Each web server will have a different mechanism for implementing SSL. In all cases, you will need to purchase a validated and signed SSL certificate from a known certificate authority (also known as a CA). Rather than cover all possible web servers and how you configure SSL, we'll cover the example application we built for this guide. You can translate these instructions for your application.

LOOK A GUIDELINE!
Encrypt all communications

Our application was developed on Node.js and we purchased a SSL certificate through GoDaddy. To purchase the SSL certificate, we first needed to create a certificate signing request (CSR). To create a CSR, we executed these commands:

```
$ openssl req -new -newkey rsa:2048 -nodes -keyout hackthis.inversoft.com.key -out
hackthis.inversoft.com.csr
```

This created two files in the current directory:

- A 2048 bit RSA private key file named `hackthis.inversoft.com.key`
- A certificate signing request file named `hackthis.inversoft.com.csr`

We used the CSR to initiate the creation of our SSL certificate with GoDaddy. Once the certificate was issued, we installed it on our Application Server in a secure location that was only readable by the application user we created above. Here are the commands we used to install the certificate after we downloaded it to our local computer:

```
$ scp * inversoft@hackthis.inversoft.com:/tmp
$ ssh inversoft@hackthis.inversoft.com
$ sudo mkdir -p /usr/local/application/ssl
$ sudo cp /tmp/hackthis.inversoft.com.* /usr/local/application/ssl
$ sudo chown -R application:application /usr/local/application
$ sudo chmod -R go-rwx /usr/local/application
```

These commands read as follows:

1. Copy SSL certificate files to the server.
2. Log into the server as the inversoft user.
3. Make a new directory for our application's SSL certificate files (the application will be deployed in this directory as well).
4. Copy the SSL certificate files to the directory.
5. Change the ownership of the directory and the files so they are owned by the application user.
6. Change permissions of the directory and files so that only the application user can access them.

The last step we performed to setup our application for SSL was to enable it in our Node.js code. We always want visitors to be redirected to the SSL version of our website so we setup a redirect from all HTTP requests. Here's the basic Node.js code that sets up SSL for our application. This code runs from `/usr/local/application` so the `ssl` directory is relative from that location:

```
var https = require('https');

var fs = require('fs');
var options = {
  key: fs.readFileSync('ssl/hackthis.inversoft.com.key'),
  cert: fs.readFileSync('ssl/hackthis.inversoft.com.crt')
};

https.createServer(options, function (req, res) {
  // Secure code goes here!
}).listen(3003);
```

```
// Redirect from http port 80 to 443 (https)
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(301, { "Location": "https://hackthis.inversoft.com" + req.url });
  res.end();
}).listen(3000);
```

5.3 Password Security

When a user registers, they will provide a password for their account. The application needs to ensure that all passwords provided by users are secure. This usually means that the password is sufficiently complex.

LOOK A GUIDELINE!
Automate security checks and constraints

NOTE: In most cases, a complex password is at least 10 characters, contains an uppercase letter, a lowercase letter, a number and a non-letter/non-number character (i.e. punctuation).

If you aren't using a third-party user database that can ensure passwords are secure, you will need to implement this password validation routine yourself. Here's some simple Java code that can be used to validate passwords:

```
public class PasswordValidator {
  public boolean validate(String password) {
    if (password == null || password.trim().length() == 0) {
      return false;
    }

    if (password.length() < 10) {
      return false;
    }

    char[] ca = password.toCharArray();
    return containsLowerCase(ca) && containsUpperCase(ca) && containsNonAlphabetic(ca);
  }

  private boolean containsLowerCase(char[] ca) {
    for (char c : ca) {
      if (Character.isLowerCase(c)) {
        return true;
      }
    }
    return false;
  }

  private boolean containsNonAlphabetic(char[] ca) {
    for (char c : ca) {
      if (!Character.isAlphabetic(c)) {
        return true;
      }
    }
    return false;
  }

  private boolean containsUpperCase(char[] ca) {
    for (char c : ca) {
      if (Character.isUpperCase(c)) {
        return true;
      }
    }
    return false;
  }
}
```

In addition to password validation, you should never store passwords in plain-text. Passwords should always be hashed using a strong, one-way hash algorithm. The hash is then stored in the database. When a user logs in, you will perform the same hashing on the password they provide on the login form and compare it with the hash in the database.

In addition to using a hashing algorithm, you should also be salting the password and performing multiple hash passes. This will prevent brute force or lookup attacks on passwords. Each user should have a different salt stored in the database and your application should perform in-memory salt modification as well. All of these techniques will make the passwords impossible to determine. A salted, in-memory modified password prior to hashing might look something like this:

```
[{SALT}]PASSWORD
```

If the salt is a UUID like “`16e49f4f-fd87-41b9-8013-57ed3b0403d8`” and the password is something complex like “Ksie923kd-A291kndj”, the initial value would look like this:

```
[{16e49f4f-fd87-41b9-8013-57ed3b0403d8}]Ksie923kd-A291kndj
```

This value is then hashed with an algorithm like SHA-256 7 times. The result in this case would be the hash:

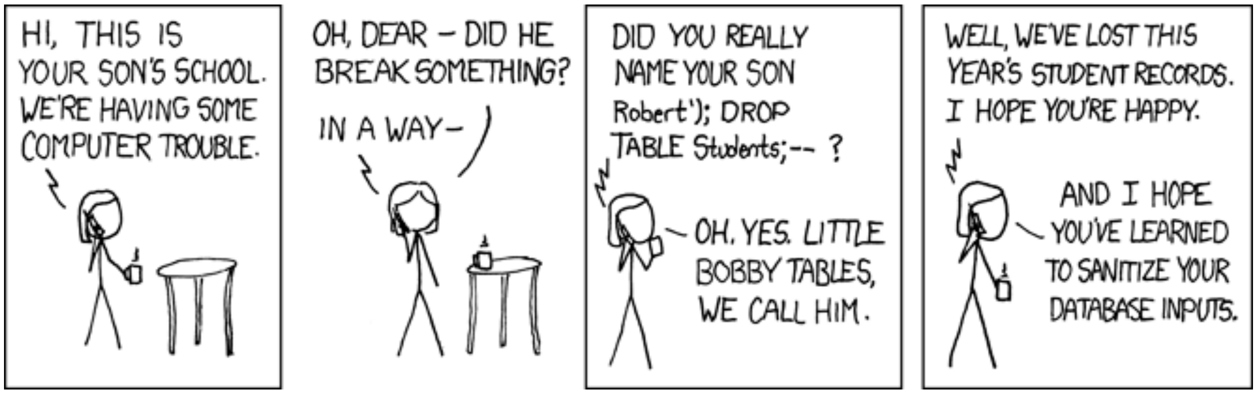
```
2206b1bbf30cf86bd1ecb952d1676aa3b9df45a9704e50e899d9615e20e53c2c
```

Here is some simple Java code that illustrates a multiple pass SHA hashing system:

```
public class MultiplePassSaltedSHA256PasswordHasher {
  public String encrypt(String password, String salt) {
    String salted =  "[{" + salt + "}]" + password;
    MessageDigest messageDigest;
    try {
      messageDigest = MessageDigest.getInstance("SHA-256");
    } catch (NoSuchAlgorithmException e) {
      throw new IllegalArgumentException("No such algorithm [SHA-256]");
    }

    byte[] hash = salted.getBytes();
    for (int i = 0; i < 7; i++) {
      hash = messageDigest.digest(hash);
    }

    return new String(Base64.getEncoder().encode(hash));
  }
}
```

5.4 SQL Injections

SQL injection is an application level vulnerability that is caused when an application builds SQL queries (or any type of database query) directly from user input. The XKCD comic strip illustrates this principle:


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

A classic example of a SQL injection is a set of search results or tabular data that can be sorted. Many applications put the sorting column and direction on the anchor tags or the select box used to select the sort. When a user selects the sort, the browser will make a request like this:

LOOK A GUIDELINE!
Cleanse user input

```
http://www.my-application.com/some-data?sort=first_name+desc
```

The application takes this information and appends it to the SQL query like this:

```
select * from users order by first_name desc
```

This works, but opens a massive security risk for the application. If a hacker changes the URL to this:

```
http://www.my-application.com/some-data?sort=(select%20CAST((select
concat(email,':',password) from users limit 1) as date))+desc
```

It will cause an SQL error, which will also contain the email and password for the first user in the database. The error might look something like this:

```
"admin@inversoft.com:173ad94aa37d140a5aab46795b14cef88efe8be76930b90460b5da60f4cff76e"
```

If the error is sent back from the server to the browser either in JSON or HTML, hackers can then modify the URL above to learn everything about the database and query the data of every user.

There are a number of ways to remove SQL vulnerabilities from your code. Here are a few:

Never send errors or exceptions back to the front-end. Instead, log them and send back a nice error message that says something like “Sorry, something bad happened. Please try again.”

Protect SQL queries by validating user input and preventing injections. This can be implemented using a simple check like this:

```
if (!sort.matches("[a-zA-Z_0-9.]+\\s+[adescADESC]+")) {
  sort = "";
}
```

This code ensures that the sort matches a specific regular expression. If it doesn’t match, it removes the sort completely.

Another possible solution is to enumerate the possible sort options and pass an enumeration value instead of the SQL snippet on the URL. To use this method, the URL above would become:

```
http://www.my-application.com/some-data?sort=first_name_DOWN
```

And the code might look like this:

```
if (sort == SortEnum.firstName_DOWN) {
  orderBy = "first_name desc";
}
```

A final solution is to use prepared statements whenever possible. Prepared statements allow you to put placeholders into a SQL statement that are replaced with a value at runtime. Most database drivers will properly escape the value that is passed into the variable which makes SQL injections impossible. An SQL statement that uses prepared statements looks like this:

```
select * from users where name = ?
```

The '?' character would be replaced and escaped by the database driver. Therefore, even if a user sent a malicious string in an attempt to perform SQL injection, their attempt would fail because the database would properly escape the string. Here's an example of a malicious string that is attempting to select the first user from the database, but the database driver has correctly escaped it:

```
select * from users where name = '\';select concat(email,\':\',password) from users limit
1;'
```

It is a good idea to centralize all of your database logic into a single module and regularly query the code for string concatenation or variable replacements that might open your application up to security risks.

5.5 Configuration

LOOK A GUIDELINE!

Keep it on the server

Server applications generally need access to a database in order to store and retrieve data. The credentials that the application uses to connect to the database should be stored in a configuration file. However, this configuration file should not be checked into source control. A development version of the configuration file can be checked in to make it simpler for developers to work. Therefore, a mechanism that determines which configuration file to load is required. Here's a simple code snippet from our Node.js application that determines if a production configuration file exists or if the development version should be used:

```
var config = require("./config-dev.json");
try {
  var stats = fs.statSync("/usr/local/application/config/config-production.json");
  if (stats.isFile()) {
    config = require("/usr/local/application/config/config-production.json");
  }
} catch (err) {
  // Already initialized to a good value, ignoring the exception for now
}

module.exports = config;
```

This will allow production configuration to be stored in a secure location on the production server and development configuration to be stored in source control for development.

It is also vital to secure the production configuration on the server. If a hacker manages to log into a server and they can simply read the application configuration file, they'll immediately have full access to the database. This is the same level of security breach as if they gained access to the database server.

Therefore, the configuration file must be secured. To secure the file on Linux simply change the ownership and permissions of the configuration directory and file. Lock down your configuration file by executing these commands as the root user:

```
$ chown -R application:application /usr/local/application/config
$ chmod 700 /usr/local/application/config
$ chmod 400 /usr/local/application/config/*
```

This will ensure that only the user running the application can read the configuration file.

All sensitive configuration should be locked down in a similar manner. This includes:

- SSL private keys
- API keys and tokens for third-party software
- Database usernames and passwords

5.6 Database Security

We talked about securing the database at the server level in the previous section. Here we will cover securing the database at the application level. Since most applications connect to a database, these connections must be secured.

The most important aspect of securing a database is the database user the application connects with. Never connect your application to your database as a superuser (AKA a root database user). Instead, create an application user and limit the privileges of this user to only those needed for the application.

LOOK A GUIDELINE!
Use the least privilege possible

For our example application, we created a database user named `inversoft`. We granted this user permission to only the database for our application. Here's the SQL statements we used:

```
mysql> create user 'inversoft'@'192.81.133.144' identified by 'super-strong-password';
mysql> grant insert, select, update, delete on users_todos.* to 'inversoft'@'192.81.133.144'
```

This creates our user and then allows the user to perform the standard SQL data options (insert, select, update and delete) on our database only.

5.7 User Data Hacks

Most modern applications use a combination of server-side code and client-side code. These two layers of code communicate with each other via APIs. Often, JSON data is sent between the client and server using AJAX requests. This new model of application development adds a lot of flexibility, but also opens up a lot of security risks.

After a user logs in, the application will need the user id in order to look up additional data for the user as well as ensure the user has the correct permissions to take actions. This user id will also be used when the user modifies, creates or deletes their data.

Since the user id is such a powerful component of an application, it needs to be secured. Applications should never send the user id to the front-end nor should the server require the user id to be sent from the client to take an action. Instead, the user id should only be stored on the server in a session that is associated with the client (i.e. the browser the user is using).

LOOK A GUIDELINE!
Keep it on the server

Here's a code snippet from our Node.js application that is leveraging the Passport user database to store the user (which has an id field) in the session:

```
var express = require("express");
var app = express();

// ...

// Setup secure session IDs that are stored in a client cookie
app.use(session({
  secret: uuid.v4(),
  name: 'sessionId',
  cookie: {
    secure: true,
    maxAge: 3600 * 1000
  }
}));

// Store the user in the session on login
var router = express.Router();
var passportClient = new Passport.PassportClient("API-KEY", "http://localhost:9011");
router.route("/login").post(function (req, res) {
  passportClient.login(req.body).then((clientResponse) => {
    req.session.user = clientResponse.successResponse.user;
    // ...
  });
});
```

Each request then uses this id when performing actions on behalf the user. Here's a code snippet of how a user id is looked up from the session to store some data in the database:

```
router.route("/todos").get(function (req, res) {
  var todos = todo.retrieveCompletedTodos(req.session.user.id);
  // ...
});
```

In addition to securing the user id, you should also be authorizing API calls from your front-end to ensure the user making the request has the correct permissions to do so. In most cases, a user will have a list of roles and your code will check those roles to ensure the user has permission to take an action. Here's an example of this check in our Node.js application:

```
router.route("/todos").post(function (req, res) {
  var user = new User(req.session.user);
  if (user.hasRole("CREATE_TODO")) {
    todo.createTodo(req.body.todo, req.session.user.id);
  }
});
```

One thing to note about this code is that the Express session only stores data, not objects. In some web server frameworks, you can store objects in the session. To get around this limitation, we create a new User object using the user data we stored in the session. This object has helper functions on it, including the Role function that allows us to check if the user has a specific role.

You can continue to secure your application on the server by adding additional checks and storing other secure data in the session.

5.8 Social Hacks

Social hacking is another issue that your application could be subject to. An example of a classic social hack is when a user gets a support representative to change the password of someone else's account so that they can log into that account.

Here's a scene from the movie Hackers that illustrates this concept:

https://www.youtube.com/watch?v=2efhrCxI4J0

Of course, you'll want to ensure your "BLT drive" is secure, but in this case you need to ensure that your application prevents your employees from falling prey to social hacks.

The best way to prevent social hacking is to ensure that employees, more specifically customer service representatives, don't have too much power. For example, customer service representatives should not be able to change user passwords. Instead, they should only be allowed to send password reset emails to users so that the user can change their own password.

6 Summary

This guide covers a set of best practices that will help you secure your user data. It is by no means exhaustive, but if you implement each of the concepts above your user data will be highly secure.

We also would like to note that this guide is applicable for both large and small companies. Even if you have 10,000 servers and large configuration, deployment and management processes, you can still use the concepts in this guide to ensure your processes are secure.

If you find any errors in this guide, please let us know. Also, if you find any vulnerabilities in this guide, send us an email at dev@inversoft.com so that we can investigate them and update the guide.

7 References

In addition to the experience and knowledge of our development team, we used additional resources to build this guide. Below is a non-exhaustive list of the references and resources we used:

Password Complexity

http://xmodulo.com/set-password-policy-linux.html

SSH

http://askubuntu.com/questions/20450/disable-root-account-in-ubuntu

IPTables

https://wiki.archlinux.org/index.php/iptables

https://www.digitalocean.com/community/tutorials/how-to-implement-a-basic-firewall-template-with-iptables-on-ubuntu-14-04

http://www.howtogeek.com/177621/the-beginners-guide-to-iptables-the-linux-firewall/

http://serverfault.com/questions/373871/dont-understand-00-iptable-syntax

http://serverfault.com/questions/157375/reject-vs-drop-when-using-iptables

http://www.thegeekstuff.com/2011/01/iptables-fundamentals/

Two-Factor Authentication

https://www.digitalocean.com/community/tutorials/how-to-protect-ssh-with-two-factor-authentication

http://serverfault.com/questions/629883/trying-to-get-ssh-with-public-key-no-password-google-authenticator-working-o

http://www.howtogeek.com/121650/how-to-secure-ssh-with-google-authenticators-two-factor-authentication/

Monit and Intrusion Detection

http://askubuntu.com/questions/179889/how-do-i-set-up-an-email-alert-when-a-ssh-login-is-successful

https://mmonit.com/wiki/MMonit/PushoverNotification

https://mmonit.com/wiki/MMonit/SlackNotification

Backup Encryption

http://askubuntu.com/questions/95920/encrypt-tar-gz-file-on-create

Node.js SSL

https://nodejs.org/api/https.html

http://stackoverflow.com/questions/22453782/nodejs-http-and-https-over-same-port

https://en.wikipedia.org/wiki/Transport_Layer_Security

Express

http://stackoverflow.com/questions/20938620/node-js-express-disable-automatic-session-creation

http://expressjs.com/en/advanced/best-practice-security.html