A Practical Example of Software Defined Security

Using Amazon Web Services,
Chef, APIs, and Ruby

October 2, 2013

Author's Note

The content in this report was developed independently of any sponsors. It is based on material originally posted on the Securosis blog but has been enhanced and professionally edited.

Copyright

Table of Contents

Automating Cloud Security Policy Compliance

Many people focus (often wrongly) on the new risks of cloud computing, but I am far more interested in leveraging cloud computing to improve security. This paper collects recent research published at securosis.com that demonstrates the power of leveraging cloud computing, APIs, and DevOps tools to improve security. It's a practical example, and feel free to contact us at info@securosis.com if you have any questions.

Overview

In this example we will automate hooking into cloud servers ('instances') and securely deploying a configuration management tool, including automated distribution of security credentials. Specifically, we will use Amazon EC2, S3, IAM, and OpsCode Chef; and configure them to handle completely unattended installation and configuration. This is designed to cover both manually launching instances and autoscaling.

With very minor modification you can use this process for Amazon VPC. With more work you could also use it for different public and private cloud providers, but in those cases the weakest link will typically be IAM. There are a few ways you can bridge that gap if necessary – I don't know them all, but I do know they exist.

First, let's define what I mean by cloud security policy compliance. That is a broad term, and in this case I am specifically referring to automating the process of hooking servers into a configuration management infrastructure and enforcing policies. By using a programmatic configuration management system like Chef we can enforce baseline security policies across the infrastructure and validate that they are in use.

For example, you can enforce that all servers are properly hardened at the operating system level, with the latest patches, and that all applications are configured according to corporate standards.

The overall process is:

- Bootstrap all new instances into the configuration management infrastructure.
- Push policies to the servers, including initial and update policies.
- Validate that policies deployed.
- Continuously scan the environment for rogue systems.
- Isolate, integrate, or remove the rogue systems.

All of this is insanely cool, and only the very basics of Software Defined Security.

Here is specifically what we will cover:

- Using cloud-init to bootstrap new Amazon EC2 instances.
- Use Amazon IAM roles to provide Temporary rotating security credentials to the instance to access the initial configuration file and digital certificate for Chef.
- Automatic installation of Chef, using the provided credentials.
- Instances will use the configuration file and digital certificate to connect to a Chef server running in EC2.
- The Chef server is locked down to only accept connections from specified Security Groups.
- S3 is configured to only allow read access of the credentials from instances with the assigned IAM role.
- The tools in use and how to configure them manually.

I will start with how IAM roles work and how to configure them, next how to lock down access using IAM and Security Groups, then how to build the cloud-init script with details on the command-line tools it installs and configures, and finally how it connects securely back up S3 for credentials.

Okay, let’s roll up our sleeves and get started…

Using Amazon IAM Roles to Distribute Security Credentials (for Chef)

As I discussed in the previous section, you can combine Amazon S3 and IAM roles to securely provision configuration files (or any other files) and credentials to Amazon EC2 or VPC instances. Here are the details.

The problem to solve

One of the issues in automating infrastructure is securely distributing security credentials and configuration files to servers that start automatically, without human interaction. You don't want to embed security credentials in the images that are the basis for these running instances (servers), and you need to ensure that only approved instances can access the credentials, without necessarily knowing anything about the instance.

The answer is to leverage the cloud infrastructure itself to identify and propagate the credentials. We can handle it all in the management plane without manually touching servers or embedding long-term credentials.

In our example we will use these to distribute an initial Chef configuration file and validation certificate.

Securely bootstrapping Chef clients

A Chef node is a server or workstation with the Chef agent running on it. This is the application that connects to the Chef server to accept configuration pushes and run scripts (recipes). There are four ways to install it on a cloud instance:

- Manually log into the instance and install the chef-client software, then transfer over the server's validation certificate and configuration file. Almost nobody does this in the cloud, outside of development and testing.
- Embed the client software and configuration files in the machine image for use at launch (instantiation). This is common but means you need to maintain your own images rather than using public ones.
- Remotely bootstrap the instance. The Chef management software (knife) can connect to the instance via ssh to configure everything automatically, but that requires its `ssh private key.
- Use cloud-init or another installation script to install Chef onto a clean 'base' (unconfigured) instance, and IAM roles to allow the instance to connect to a repository such as S3 to download the initial configuration file and server certificate. This is what I will explain.

Configuring AWS IAM roles to distribute credentials

Amazon recently added a feature called IAM roles. Amazon Web Services (like some other public and private cloud platforms) supports granular identity management down to the object level. This is critical for proper segregation and isolation of cloud assets. AWS previously only supported users and groups, which are (and I'm simplifying) static collections of users, servers, and other objects in AWS.

Users and groups are great, but they provide users or servers with static security credentials such as anAccess Key and Secret Key, X.509 certificate, or a username and password for web UI access.

IAM Roles are different. They are temporary credentials applied to AWS assets, such as a running instance. They include an Access Key and Secret Key provided to the object via an API call, and a token. You need all three to sign requests, and they are rotated (approximately every 24 hours in my experience).

So if someone steals the keys they won't work without the token. If they also get a token it expires in a day.

In our example we will create an S3 bucket to hold our Chef configuration client.rb configuration file and validation.pemdigital certificate. We will then switch over to AWS IAM to create a new role and assign it read privileges to S3. Then we will tweak the policy to only allow access to that bucket.

Finally we will launch an instance, assign the role, then log in and show the credentials. You wouldn't do this in production, but it illustrates how roles work.

Step by step

I assume you have some knowledge of AWS here. If you want granular instructions, take our class. I also assume you have a Chef server set up in EC2 someplace, or use Hosted Chef. If you want to know how to do that, take the class. :)

AWS Console

1. Log in and ensure your Chef server has its own Security Group.

2. Create a new Security Group for your instances (or pick any group you already have). Our example is very locked down, which may not be appropriate for your environment.

3. Open ports 4000, 4040, and 80 in the Chef Server security group from your instance security group. I haven't had time to play with it, but we might be able to double down and allow access by role. I will test before Black Hat – it doesn't take long, but I just got the idea. Return the favor and open 4000 and 4040 into the instance group from the server group.

Amazon S3 section of AWS Console

1. Create a new bucket (e.g., cloudsec). Load a random file for testing later if you want. If you have a Chef server placeclient.rb and validation.pem here – you will need these to complete our example.

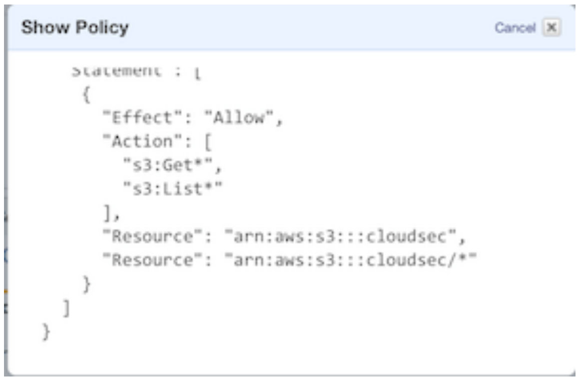
IAM section

1. Create a new role called ChefClient. You can do this all via API or by write the policy by hand, but we use the GUI.

2. Select AWS Service Roles, then Amazon EC2. This grants the designated rights to EC2 assets with the assigned role. Continue.

3. Select Select Policy Template and then Amazon S3 Read Only Access. Continue.

4. After this you can name the policy, then adjust it to apply to only the single bucket – not your entire Amazon S3 account. Change the entry “Resource: *” to “Resource: arn:aws:s3:::your_bucket”. I also added a safety wildcard, so your policy should look like the screenshot below.

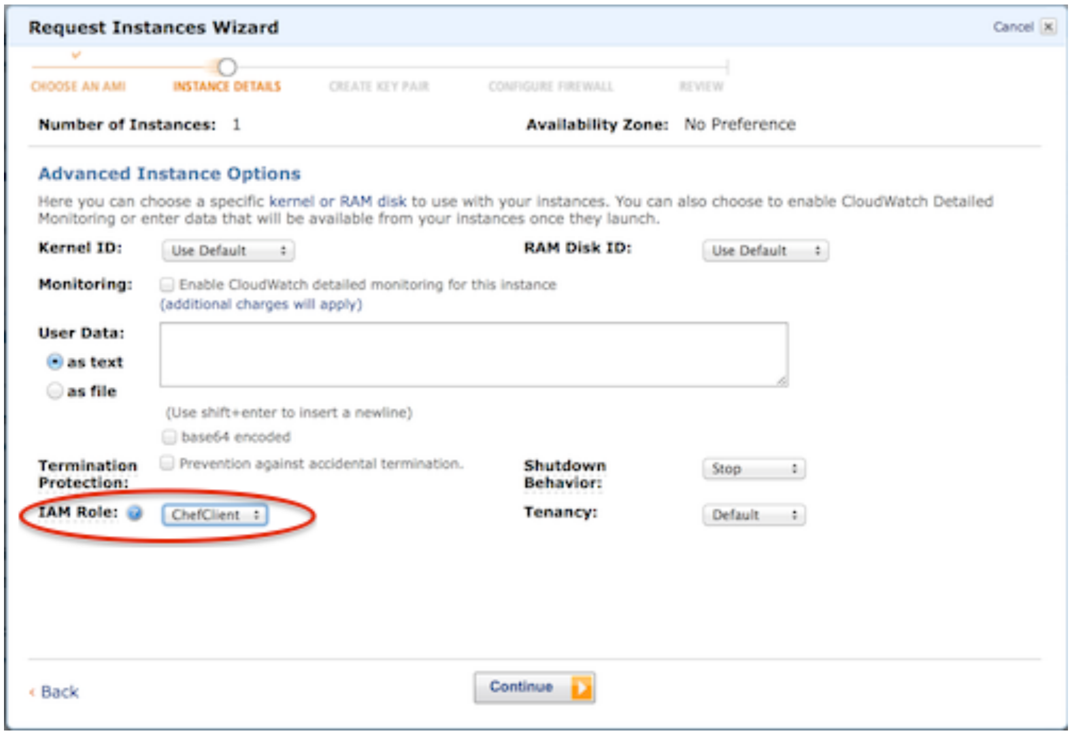


```
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "s3:Get*",
          "s3:List*"
        ],
        "Resource": "arn:aws:s3:::cloudsec",
        "Resource": "arn:aws:s3:::cloudsec/*"
      }
    ]
  }
```

EC2 section

1. Launch a new instance. Ubuntu is a safe bet, and what we use to demonstrate the temporary credentials.

2. On the Instance Details screen assign your IAM role. You also probably want to put it in the same availability zone as your Chef server, and later on into the right security group.



Number of Instances: 1
Availability Zone: No Preference

Advanced Instance Options

Here you can choose a specific kernel or RAM disk to use with your instances. You can also choose to enable CloudWatch Detailed Monitoring or enter data that will be available from your instances once they launch.

Kernel ID: Use Default
RAM Disk ID: Use Default
Monitoring: Enable CloudWatch detailed monitoring for this instance (additional charges will apply)
User Data: [✓ as text] as file
(Use shift+enter to insert a newline)
base64 encoded
Termination Protection: Prevention against accidental termination.
Shutdown Behavior: Stop
IAM Role: ChefClient
Tenancy: Default

Instance

1. Once everything is running, log into your instance.

2. Type `wget -O -- -q 'http://169.254.169.254/latest/meta-data/iam/security-credentials/myrole'`, replacing 'myrole' with the name of your role (case sensitive). You should see your temporary AWS credentials, when they were issued, and when they expire.

You have now configured your environment to support transfer of the security credentials only to instances assigned the appropriate role (ChefClient in my case). Your instance has temporary credentials that Amazon rotates for you, minimizing exposure. AWS also requires a token so the Access Key and Secret Key won't work by themselves. Our next section will show how to use cloud-init to install the Chef client software, download the configuration file and server certificate, and run a scripted install of Chef.

With this all configured, all you need to do is assign the role and embed the cloud-init script, and instances will automatically (and securely) configure themselves and connect to the Chef server.

Using cloud-init and s3cmd to Automatically Download Chef Credentials

The previous section described how to use Amazon EC2, S3, and IAM as a framework to securely and automatically download security policies and credentials. That's the infrastructure side of the problem, and this post will show what you need to do to the instance to connect to this infrastructure, grab the credentials, install and configure Chef, and connect to the Chef server.

The advantage of this structure is that you don't need to embed credentials into your machine image, and you can use stock (generic) operating systems are on public clouds. In private clouds it is also useful because it reduces the number of machine images to maintain.

These instructions can be modified to work in other cloud platforms, but your mileage will vary. They also require an operating system that supports cloud-init (Windows uses ec2config, which I know very little about, but also appears to support user data scripts).

I will walk through the details of how this works, but you won't use any of these steps manually. They are just explanation, to give you what you need to adapt this for other circumstances.

Using cloud-init

cloud-init is software for certain Linux variants that allows your cloud controller to pass scripts to new instances as they are launched from an image (bootstrapped). It was created by Canonical (the Ubuntu guys) and is very frequently packaged into Linux machine images (AMIs). ec2config offers similar functionality for Windows.

Users pass the script to their instances, specifying the User Data field (for web interface) or argument (for command line). It is a bit of a pain because you don't get any feedback – you need to debug from the system log – but it works well and allows tight control. Commands run as root before anyone can even log into the instance, so cloud-init is excellent for setting up secure configurations, loading ssh keys, and installing software.

Note that cloud-init is a bootstrapping tool for configuring an instance the first time it runs – it is not a management tool because after launch you cannot access it any more.

For an example see our full script at the bottom of this post.

You can download and manipulate files easily with cloud-init, but unless you want to embed static credentials in your script there is an authentication issue. That's where AWS IAM roles and S3 help, thanks to a very recent update to s3cmd.

Configuring s3cmd to use IAM roles

s3cmd is a command-line tool to access Amazon S3. Amazon S3 isn't like a normal file share – it is only accessible through Amazon's API. s3cmd provides access to S3 like a local directory, as well as administration of S3. It is available in most Linux repositories for packaged installation, but the bundled versions do not yet support IAM roles. Version 1.5 alpha 2 and later add role support, so that's what we need to use.

You can download the alpha 3 release, but if you are reading this post in the future I suggest checking for a more recent version on the main page, linked above.

To install s3cmd just untar the file. If you aren't using roles you now need to configure it with your credentials. But if you have assigned a role, s3cmd should work out of the box without a configuration file.

Unfortunately I discovered a lot of weirdness once I tried to out it in a cloud-init script. The issue is that running it under cloud-init runs it as root, which changes s3cmd's behavior a bit. I needed to create a stub configuration file without any credentials, then use a command-line argument to specify that file.

Here is what the stub file looks like:

```
[default]
access_key =
secret_key =
security_token =
```

Seriously, that's it.

Then you can use a command line such as:

```
s3cmd --config /s3cmd-1.5.0-alpha3/s3cfg ls s3://cloudsec/
```

Where s3cfg is your custom configuration file (you can see the path there too).

That's all you need. s3cmd detects that it is running in role mode and pulls your IAM credentials if you don't specify them in the configuration file.

Scripted installation of the Chef client

The Chef client is very easy to install automatically. The only tricky bit is the command-line arguments to skip the interactive part of the install; then you copy the configuration files where they are needed.

The main instructions for package installation are in the Chef wiki. You can also use the omnibus installer, but packaged installation is better for automated scripting. The Chef instructions show you how to add the OpsCode repository to Ubuntu so you can "apt-get install".

The trick is to point the installer to your Chef server, using the following code instead of a straight "apt-get install chef-client":

```
echo "chef chef/chef_server_url string http://your-server-IP:4000" \
| sudo debconf-set-selections && sudo apt-get install chef -y --force-yes
```

Then use s3cmd to download client.rb & validation.pem and place them into the proper locations. In our case this looks like:

```
s3cmd --config /s3cmd-1.5.0-alpha3/s3cfg --force get s3://cloudsec/client.rb /etc/chef/
client.rb
```

```
s3cmd --config /s3cmd-1.5.0-alpha3/s3cfg --force get s3://cloudsec/validation.pem /etc/chef/
validation.pem
```

That's it!

Tying it all together

The process is really easy once you set this up, and I went into a ton of extra detail. Here's the overview:

1. Set up your S3, Chef server, and IAM role as described in the previous post.
2. Upload client.rb and validation.pem from your Chef server into your bucket. (Execute "knife client ./" to create them).
3. Launch a new instance. Select the IAM Role you set up for Chef and your S3 bucket.
4. Specify your customized cloud-init script, customized from the sample below, into the User Data field or command-line argument. You can also host the script as a file and load it from a central repository using the include file option.
5. Execute chef-client.
6. Profit.

If it all worked you will see your new instance registered in Chef once the install scripts run. If you don't see it check the System Log (via AWS – no need to log into the server) to see where you script failed.

This is the script we will use for our training, which should be easy to adapt.

```
#cloud-config

apt_update: true

#apt_upgrade: true

packages:
 -- curl

 fixroutingsilliness:
- &fix_routing_silliness |
   public_ipv4=$(curl -s http://169.254.169.254/latest/meta-data/public-ipv4)
   ifconfig eth0:0 $public_ipv4 up

   configchef:
   -- &configchef |
   echo "deb http://apt.opscode.com/ precise-0.10 main" | sudo tee /etc/apt/sources.list.d/
opscode.list
   apt-get update
   curl http://apt.opscode.com/packages@opscode.com.gpg.key | sudo apt-key add -
   echo "chef chef/chef_server_url string http://ec2-54-218-102-48.us-
west-2.compute.amazonaws.com:4000" | sudo debconf-set-selections && sudo apt-get install chef
-y --force-yes
   wget http://sourceforge.net/projects/s3tools/files/s3cmd/1.5.0-alpha3/s3cmd-1.5.0-
alpha3.tar.gz
   tar xvfz s3cmd-1.5.0-alpha3.tar.gz
   cd s3cmd-1.5.0-alpha3/
   cat >s3cfg <<EOM
   [default]
   access_key =
   secret_key =
   security_token =
   EOM
   ./s3cmd --config /s3cmd-1.5.0-alpha3/s3cfg ls s3://cloudsec/
   ./s3cmd --config /s3cmd-1.5.0-alpha3/s3cfg --force get s3://cloudsec/client.rb /etc/chef/
client.rb
   ./s3cmd --config /s3cmd-1.5.0-alpha3/s3cfg --force get s3://cloudsec/validation.pem /etc/
chef/validation.pem
   chef-client

   runcmd:
   -- [ sh, -c, *fix_routing_silliness ]
   -- [ sh, -c, *configchef]
   -- touch /tmp/done
```

Software Defined Security with AWS, Ruby, and Chef

So far we discussed automating cloud security configuration management by taking advantage of DevOps principles and properties of the cloud. Now I will build on that to show you how the management plane can make security easier than traditional infrastructure with a little ruby code.

Abstraction enhances management

People tend to focus on multitenancy, but the cloud's most interesting characteristics are abstraction and automation. Separating our infrastructure from the physical boxes and wires it runs on, and adding a management plane, gives us a degree of control that is difficult or impossible to obtain by physically tracing all those wires and walking around to the boxes.

Dev and ops guys really get this, but we in security haven't all been keeping up – not that we are stupid, but we have different priorities.

That management plane enables us to do things such as instantly survey our environment and get details on every single server. This is an inherent feature of the cloud, because if you can't find a server the cloud doesn't know where it is – which would mean a) it effectively doesn't exist, and b) you cannot be billed for it. There ain't no Neo hiding away in AWS or OpenStack.

For security this is very useful. It makes it nearly impossible for an unmanaged system to hide in your official cloud (although someone can always hook something in somewhere else). It also enables near-instant control.

For example, quarantining a system is a snap. With a few clicks or command lines you can isolate something on the network, lock down management plane access, and lock out logical access. We can do all this on physical servers, but not as quickly or easily.

(I know I am skipping over various risks, but we have covered them before and they are fodder for future posts).

In this example I will show you how 40 lines of commented Ruby (just 23 lines without comments!) can scan your cloud and identify any unmanaged systems.

Finding unmanaged cloud servers with AWS, Chef, and Ruby

This examples is actually super simple. It is a short Ruby program that uses the Amazon Web Services API to list all running instances. Then it uses the Chef API to get a list of managed clients from your Chef server (or Hosted Chef). Compare the list, find any discrepancies, and profit.

This is only a basic proof of concept – I found seen far more complex and interesting management programs using the same principles, but none of them written by security professionals. So consider this a primer. (And keep in mind that I am no longer a programmer, but this only took a day to put together).

There are a couple constraints. I designed this for EC2, which limits the number of instances you can run. Nearly the same code would work for VPC, but while I run everything live in memory, there you would probably need a database to run this at scale. This was also built for quick testing, and in a real deployment you would want to enhance the security with SSL and better credential management. For example, you could designate a specific security account with IAM credentials for Amazon Web Services that only allows it to pull instance attributes but not initiate other actions.

You could even install this on an instance inside EC2 using IAM roles, as we discussed previously.

Lastly, I believe I discovered two different bugs in the Ridley gem, which is why I have to correlate on names instead of IP addresses – which would be more canonical. That cost me a couple hours of frustration.

Here is the code. To use it you need a few things:

- An access key and secret key for AWS with rights to list instances.
- A Chef server, and a client and private key file with rights to make API calls.
- The aws-sdk and ridley Ruby gems.
- Network access to your Chef server.

Remember, all this can be adapted for other cloud platforms, depending on their API support.

```
 # Securitysquirrel proof of concept by rmogull@securosis.com
 # This is a simple demonstration that evaluates your EC2 environment and identifies instances
not managed with Chef.
 # It demonstrates rudimentary security automation by gluing AWS and Chef together using APIs.

 # You must install the aws-sdk and ridley gems. ridley is a Ruby gem for direct Chef API
access.

require "rubygems"
require "aws-sdk"
require "ridley"

 # This is a PoC, so I hard-coded the credentials. Fill in your own, or adjust the
     program to use a configuration file or environment variables. Don't forget to specify the
region...

AWS.config(access_key_id: 'your-access-key', secret_access_key: 'your-secret-key', region:
'us-west-2')
```

```
 # Fill in the ec2 class
ec2 = AWS.ec2 #=> AWS::EC2
ec2.client #=> AWS::EC2::Client

 # Memoize is an AWS function to speed up collecting data by keeping the hash in local cache.
      This line creates a list of EC2 private DNS names, which we will use to identify nodes
in Chef.
instancelist = AWS.memoize { ec2.instances.map(&:private_dns_name) }

 # Start a ridley connection to our Chef server. You will need to fill in your own credentials
or
        pull them from a configuration file or environment variables.
ridley = Ridley.new(
  server_url: "http://your.chef.server",
  client_name: "your-client-name",
  client_key: "./client.pem",
  ssl: { verify: false }
  )

   # Ridley has a bug, so we need to work on the node name, which in our case is the same as
the EC2 private DNS name.
          For some reason node.all doesn't pull IP addresses (it should) which we would prefer
to use.
  nodes = ridley.node.all
  nodenames = nodes.map { |node| node.name }

 # For every EC2 instance, see if there is a corresponding Chef node.

  puts ""
  puts ""
  puts "Instance => managed?"
  puts ""
  instancelist.each do |thisinstance|
  managed = nodenames.include?(thisinstance)
  puts " #{thisinstance} #{managed} "
  end
```

Where to go next

If you run the code above you should see output like this:

```
Instance => managed?

 ip-172-xx-37-xxx.us-west-2.compute.internal true
 ip-172-xx-37-xx.us-west-2.compute.internal true
 ip-172-xx-35-xxx.us-west-2.compute.internal true
 ip-172-3xx1-40-xxx.us-west-2.compute.internal false
```

That is the internal DNS name of the instance and whether or not it is managed by Chef. You could easily adapt this to pull different attributes, such as the instance ID.

No big deal, but think about where you could take it from here. Using the same framework, you could then:

- Identify the owner of the server.
- Quarantine it on the network.
- Transfer management within AWS to the security team.
- Snapshot it and shut it down.
- Bootstrap it into Chef automatically, if you have the private key for the server.
- Allow it to continue running but cut off external network connections.
- Automatically kick off a vulnerability assessment.

Can we do this with traditional infrastructure? Sure, by building a scanner and hoping we have the proper network access. Definitely not as easy as here, nor as automatic. This is what we call Software Defined Security.

The most interesting part of this is bridging the divide between DevOps and security. We in security are, for the most part, really behind on understanding today's operational models. Te DevOps folks are implementing a hell of a lot of security without us. But they don't understand security principles as well as we do, so they need our help.

I am learning a lot as I go through all this, and the more I learn, the more it confirms that the future of security will look very different than the way we practice it today, in very positive ways.

Who We Are

About the Author

Rich Mogull, Analyst and CEO

Rich has twenty years of experience in information security, physical security, and risk management. He specializes in data security, application security, emerging security technologies, and security management. Prior to founding Securosis, Rich was a Research Vice President at Gartner on the security team where he also served as research co-chair for the Gartner Security Summit. Prior to his seven years at Gartner, Rich worked as an independent consultant, web application developer, software development manager at the University of Colorado, and systems and network administrator. Rich is the Security Editor of TidBITS, a monthly columnist for Dark Reading, and a frequent contributor to publications ranging from Information Security Magazine to Macworld. He is a frequent industry speaker at events including the RSA Security Conference and DefCon, and has spoken on every continent except Antarctica (where he's happy to speak for free — assuming travel is covered).

About Securosis

Securosis, L.L.C. is an independent research and analysis firm dedicated to thought leadership, objectivity, and transparency. Our analysts have all held executive level positions and are dedicated to providing high-value, pragmatic advisory services.

We provide services in four main areas:

- Publishing and speaking: Including independent objective white papers, webcasts, and in-person presentations.
- Strategic consulting for end users: Including product selection assistance, technology and architecture strategy, education, security management evaluations, and risk assessments.
- Strategic consulting for vendors: Including market and product analysis and strategy, technology guidance, product evaluations, and merger and acquisition assessments.
- Investor consulting: Technical due diligence including product and market evaluations, available in conjunction with deep product assessments with our research partners.

Our clients range from stealth startups to some of the best known technology vendors and end users. Clients include large financial institutions, institutional investors, mid-sized enterprises, and major security vendors.

Securosis has partnered with security testing labs to provide unique product evaluations that combine in-depth technical analysis with high-level product, architecture, and market analysis.