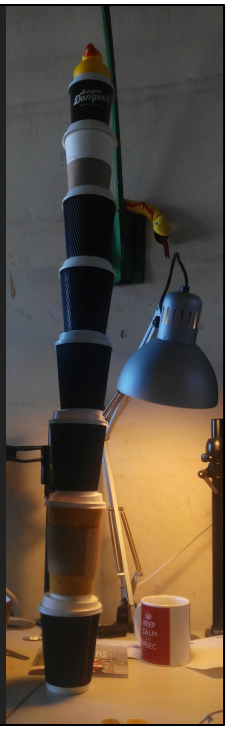
Full-mesh IPsec network

10 Dos and 500 Don’ts

$ whoami

- Fran Garcia
- SRE @hostedgraphite
- “Break fast and move things”
- Absolutely no networking/cryptography background
- No, seriously, totally unqualified to give this talk



What this talk is not

A success story

An introduction to IPsec

A HOWTO

A set of best practices

This talk is not going to be about a success story… Even if we got sort of got there in the end.
It's not going to be an introduction to IPsec, but we'll try to briefly introduce some of the basic concepts. IPsec is too vast a topic to cover in this format, so we'll just go for the "quick and mostly incorrect" here.
It's certainly not a HOWTO, but we'll share part of our configuration.
And it's not going to be a set of best practices, but there will be a couple of worse practices thrown in.

What we'll talk about

Hosted Graphite pre-IPsec

What's this IPsec thing anyway and why should I care?

Hosted Graphite does IPsec!

Everything we did wrong (well, the least embarrassing bits)

Here are the things we'll talk about.
I want to talk a little bit about where Hosted Graphite was before we started using IPsec, what we were using for our VPN needs and why we needed to change.
we'll try to get an idea of what IPsec is and why it could be useful to you, we'll introduce some of the basic concepts enough to get a feel of where some of the pitfalls and strengths are.

Then we'll talk about our project to roll out IPsec in migration and I'll try to share some of the lessons learned sprinkled with a little bit of pain and misery.

TL;DW

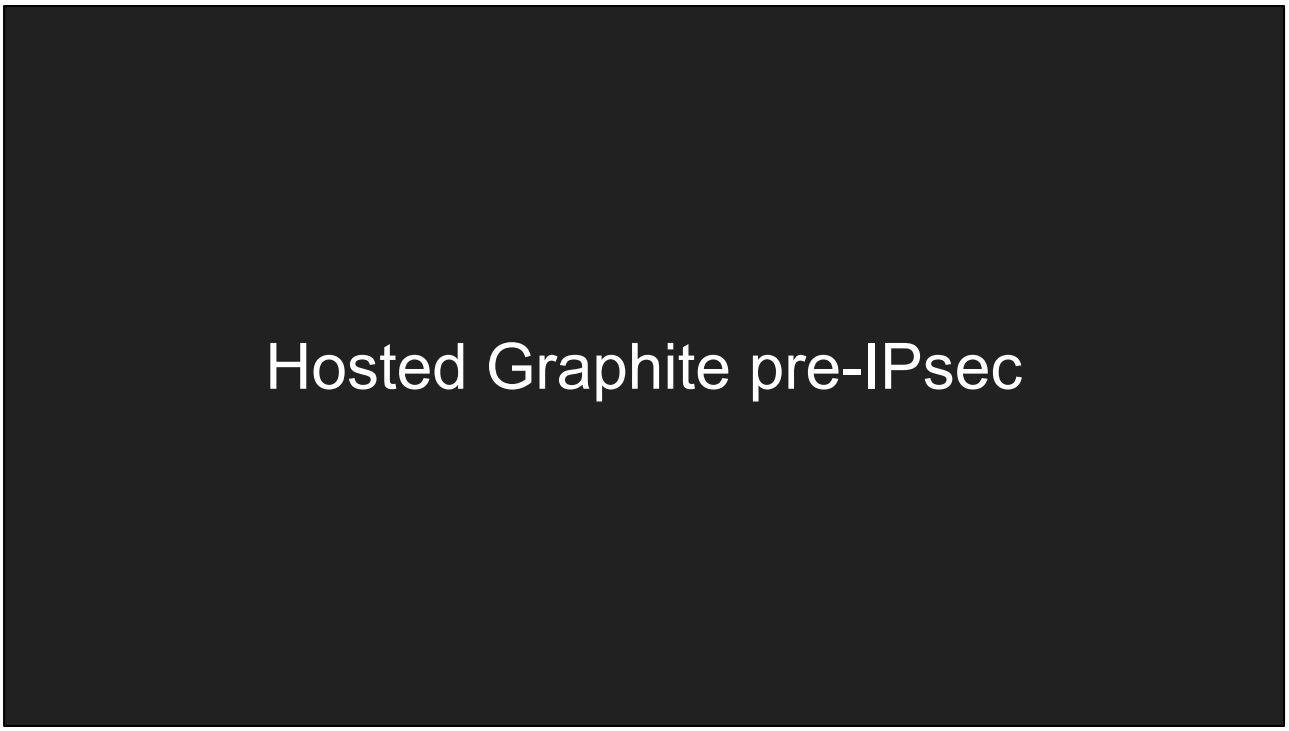
“IPsec is awful”

(trust me on this one)

Turns out IPsec is pretty awful! Then again, in terms of IP security protocols it’s probably the best we have at the moment.

Probably the main problem with IPsec as we’ve experienced it is a combination of overwhelming complexity of the standard coupled with very little documentation or experiences coming from people using it in a non-traditional way, on a cloud environment or full-mesh across different providers.

That’s a bad combination and one of the main motivations for this talk.


Hosted Graphite pre-IPsec

In the beginning, there was n2n...

Early days at Hosted Graphite:

- A way to secure communications for riak was needed
- Not many servers to maintain

Enter n2n:

- P2P VPN software
- Supports compression and encryption
- Really easy to setup and maintain

When I joined Hosted Graphite in June 2015, our internal network ran on a peer to peer vpn software called n2n.

One of the main virtues of n2n is how easy it is to set up, there's no need to maintain the list of nodes that belong to your cluster and it will transparently deal with things like NAT and broadcasting for you.

During Hosted Graphite's early days n2n it was a good choice to secure internal communications across a fairly small set of servers.

Wait, so what's the catch?

Best description from an HG engineer: "academic abandonware"

Relies on a central node (supernode):

- No supernode, no network

Single-threaded, not really efficient:

- Became a bottleneck, increasing latency for some of our services

Initially configured on a /24 private IP space

- We were running out of IP addresses!

Unfortunately, that ease of use is where n2n's advantages both begin and end, and as both our traffic and our server count started to grow, n2n was proving less and less attractive for us.
N2n relies on a "supernode" not only for discovering other nodes, all edge nodes will flat out refuse to communicate if the supernode goes down, which creates a massive single point of failure.

The edge process is also single-threaded, and not incredibly efficient, becoming the cause of high latencies in some of our services.

As n2n emulates a LAN, during the initial rollout hosts were given IP addresses in a /24 private IP space, and we realised we were running out of IP addresses in that space, which meant that we would not be able to continue scaling horizontally when hitting that threshold. At some given point we had roughty less than 10 available IP addresses on that network.

Replacing n2n

Our requirements:

- Can’t depend on fancy networking gear
- Cluster spans multiple locations/providers
- We don’t trust the (internal) network!
- Must be efficient enough not to become a bottleneck!
- Simple security model (no complex/dynamic firewall rules)
- Can be implemented reasonably quickly

So we needed to get off n2n and we needed to get off it fast, so we started gathering requirements.

One of our requirements was that our internal VPN shouldn’t limit in any meaningful way our choice of a hosting provider. We should be able to host servers in both bare metal and public clouds without our VPN being an issue.

Given that variety of providers we can’t assume we have access to the networking gear, or even trust the network, we don’t know what network devices are sitting between any given two hosts, so we assume they all have an “NSA approved” sticker in them.

Ideally our new system would be efficient enough so it wouldn’t become a bottleneck for us. We can tolerate some overhead, but our VPN shouldn’t be severely limiting the amount of traffic any given server can handle.

We’re a small company and have neither a dedicated security team or a network team, so it’s important for us that our security model is simple, the less different things we need to secure the better chance we’ll have at doing it right, so, for example, a solution that doesn’t require us to muck about with complex firewall rules that are easy to get wrong is a plus.

We also mentioned how we were under time constraints to identify and roll out a replacement for n2n, which means that we’d need to be reasonably confident that we

would be able to perform the migration in a reasonable amount of time, mainly disqualifying any options requiring significant development time on our part.

In a similar vein, given the many services we run, retrofitting encryption at the application layer is not really a choice. The more services we run the greater our chances of getting it wrong for one given service, which will reduce the overall security of the whole system. That also includes third-party software that we can't actually protect without incurring big overhead costs in both performance and complexity (like using stunnel in front of every service)

Potential n2n alternatives

We looked at a bunch of possible alternatives and most of them:

- Were not really designed for a full-mesh network (OpenVPN)
- Encrypt data in user space, incurring a performance penalty (tinc)
- Would tie us to a single provider (like AWS VPCs)
- Involve modifying and rearchitecting all our services
  - (rolling our own application layer encryption)

So after analyzing all our options IPsec won… almost by default

We evaluated a bunch of alternatives and didn't quite find anything that fit the bill given our requirements. We needed something we could use efficiently in a full-mesh architecture and none of the alternatives seemed to quite achieve that.

IPsec for the super-impatient

Let’s have a super quick and mostly incorrect tour of IPsec, just enough to get familiar with some of the concepts.

So what's this IPsec thing anyway?

Not a protocol, but a protocol suite

Open standard, which means lots of options for everything

66 RFCs linked from wikipedia page!

So what is IPsec?

IPsec is a protocol suite that secures communication between hosts at the IP level.

It's an open standard, which means you'll drown in choices and RFCs.

It is highly flexible, which unfortunately means it's also really complex.

What IPsec offers

At the IP layer, it can:

- Encrypt your data (Confidentiality)
- Verify source of received messages (Data-origin authentication)
- Verify integrity of received messages (Data Integrity)

Offers your choice of everything to achieve this

IPsec can be used to encrypt transmitted data.
It can also be used to provide data-origin authentication, allowing you to authenticate the source of the packets received, and data integrity. That is, to verify that a given message hasn't been tampered with.

Depending on your choice of protocol you'll be able to achieve all of this, some of this or none of this.

Choices, choices everywhere

What protocol?

- Authentication Header (AH): Just data integrity/authentication*
- Encapsulating Security Payload (ESP): Encryption + integrity/auth (optional)
- AH/ESP

(TL;DR - You probably want ESP)

*Legend says AH only exists to annoy Microsoft

So from the start you're faced with two choices, what protocol do I use? In what mode should it run?

Our first option is what protocol to use.
We have two choices here, the first one being authentication header.
AH does not actually encrypt any data, so it can't be used to provide confidentiality on its own.

What it does provide is data-origin authentication and data integrity for the full IP packet, including the header.

Then we also have ESP.
ESP does provide confidentiality and has the option to also provide authentication and integrity, but just for the IP payload, not the header.
You can technically use ESP with the so-called NULL encryption which means that no encryption will be performed

So which one should I use?

Given that most people using IPsec are interested in encryption you're probably going to want to use ESP. If you want to ensure integrity on the IP header as well as the payload you can actually combine both AH and ESP, but for most cases ESP will be more than enough, and with IPsec the less complexity we add the better.

Then you’ll need to decide on encryption/hashing algorithm and a thousand other things.

Second choice... Tunnel or Transport mode?

	Encapsulates header	Encapsulates payload	Works for host-to-host	Works for site-to-site
Tunnel Mode	YES	YES	YES	YES
Transport Mode	NO	YES	YES	NO

*Transport mode might incur in a slightly smaller overhead and be a bit simpler to set up

Ok, so we didn't find the expert consensus we had hoped, let's take a look at our options.

On transport mode we only encrypt/authenticate the payload of the IP packet, leaving the routing intact. That means there's no confidentiality or integrity guarantees on the header itself. This mode is often useful for host-to-host networks.

On tunnel mode the whole packet is encrypted/authenticated, including the header. The whole packet will be encapsulated and a new header will be added to it.
It's good for connecting two networks together while at the same time hiding details of your internal network from the IP header.

If you're going to be using IPsec to connect two gateways you'll want to use tunnel mode.
Transport mode is probably only an option for a host-to-host network.

In our particular case we ended up going with transport mode as it seemed to be the simpler solution, but it wouldn't have made a big difference either way. Aside from the fact that we haven't tested it extensively, we could migrate to tunnel mode tomorrow with a minimal configuration change.

For a new rollout I would probably recommend tunnel mode unless you really want to keep overhead to a minimum.

IPsec: What's a SP (security policy)?

Consulted by the kernel when processing traffic (inbound and outbound)

```
"From host A to host B use ESP in transport mode"

"From host C to host D's port 443 do not use IPsec at all"
```

Stored in the SPD (Security Policy Database) inside the kernel

In order to define when and how the kernel should use IPsec we rely on security policies. The security policies are a way to tell the kernel that a given traffic flow should be secured by IPsec in tunnel/transport mode using a given algorithm, or not protected at all, or be dropped.

These are stored in what's known as the SPD or SPDB inside the kernel.

IPsec: What's a SA (Security Association)?

Secured unidirectional connection between peers:

- So need two for bidirectional communication (hosta->hostb, hostb->hosta)

Contains keys and other attributes like its lifetime, IP address of peer...

Stored in the SAD (Security Association Database) inside the kernel

The Security Association is probably the central concept of IPsec. It defines *how* two hosts will manage to communicate securely. That means that when establishing a Secure Association both hosts will define the algorithms and the session keys they will use for this communication. All this information forms the SA.

The SA itself is identified by a triplet: being a unique ID for this SA, the IP address of the peer, and what security protocol to use (AH or ESP)

All SAs live inside the SADB inside the kernel.

IKE? Who's IKE?

"Internet Key Exchange"

Negotiate algorithms/keys needed to establish secure channel between peers

A key management daemon does it in user space, consists of 2 phases

So what happens when a host decides it wants to use IPsec to talk to another host? You can define your associations and keys manually, which depending on your environment and scale may or may not be practical, but chances are you'll want your hosts to dynamically figure this out for you.
They'll need to agree in a set of algorithms and keys to use to establish this connection, and that's where IKE comes in.

IKE is usually performed by a key management daemon in user space, while the actual data encryption/transmission is performed by the kernel.

IKE consists on two phases...

IPsec: IKE Phase 1

Lives inside the key management daemon (in user space)

Hosts negotiate proposals on how to authenticate and secure the channel

Negotiated session keys used to establish actual (multiple) IPsec SAs later

First we have phase 1:

During phase 1 both peers authenticate each other based on pre-shared keys or certificates and perform a key exchange to establish session keys.
This session will get renegotiated when its negotiated lifetime expires.

The associations formed here, also known as IKE SAs do not live inside the kernel, just inside the key management daemon.

Once both hosts have agreed on how to secure this session and established the session keys they will be used to establish the actual IPsec associations during phase 2

IPsec: Phase 2

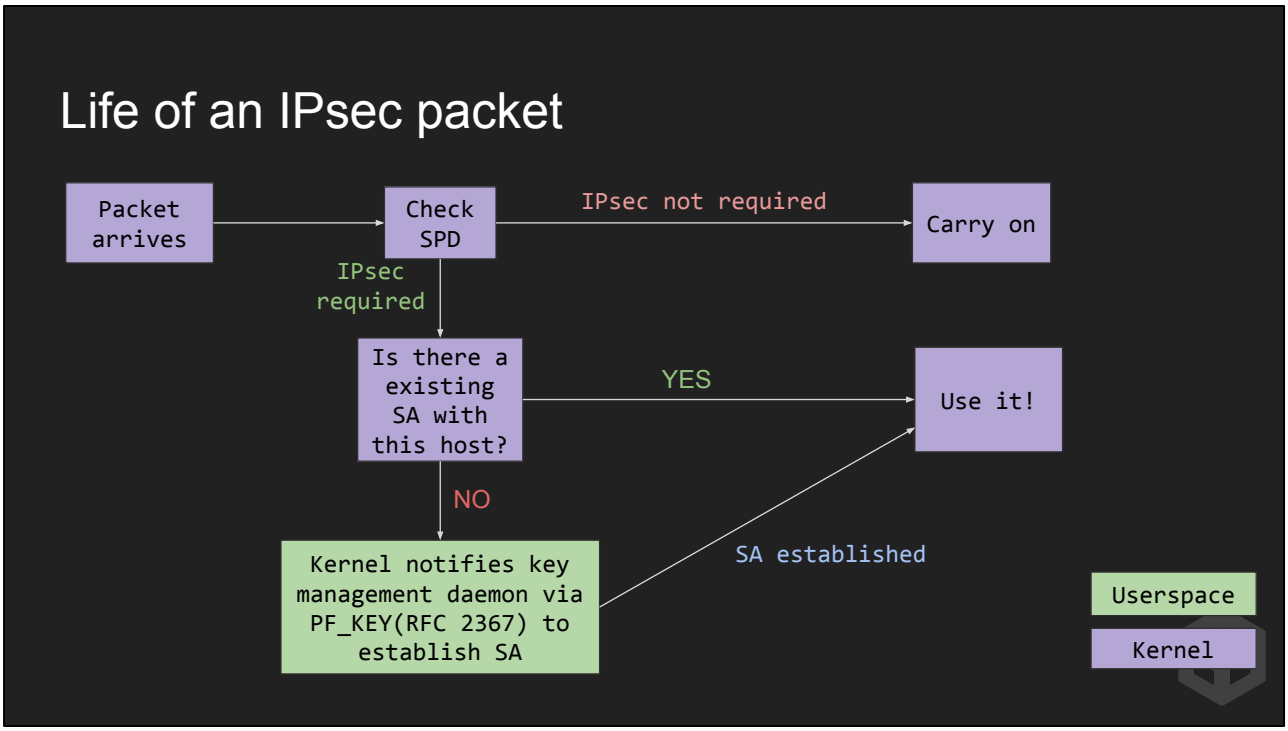
Negotiates IPsec SA parameters (protected by IKE SA) using phase 1 keys

Establishes the actual IPsec SA (and stores it in SADB)

Can renegotiate when close to end of lifetime

During phase 2 is when the actualy IPsec associations are formed. Using the keys established during phase 1 both hosts will negotiate the algorithms/parameters they will use to secure communication. When phase 2 is completed the actual IPsec Security association is formed and stored in the kernel.

When close to the end of its agreed-upon lifetime the key management daemon will be signaled by the kernel and a new SA will be negotiated using the already existing phase 1.



So here's a rough breakdown of what happens when a packet arrives (Basically the same workflow applied when we're about to send one)

The kernel will check its SPD and see if any security policies mention this specific traffic flows. If it's not mentioned or the policy explicitly says IPsec it's not needed then the packet will be processed normally.
If IPsec is required, the kernel will now check the SADB for the presence of a suitable Security Association with that other host. If one is found it will be used.
If no suitable Security Association exists the key management daemon will be notified via PF_KEY so it can perform the key exchange and establish a SA with the peer. Once that's done we can use it for any further communication attempts between hosts.

Some helpful commands

`ip xfrm` is pretty powerful. Some basics:

```
$ ip xfrm policy            # Dump the contents of the SPDB

$ ip xfrm state             # Dump the contents of the SADB

$ ip xfrm monitor           # Dump all changes to SADB and SPDB as they happen

$ ip xfrm state flush       # Flush all state in the SADB (dangerous!)
```

Documentation is... not great: http://man7.org/linux/man-pages/man8/ip-xfrm.8.html

Most of the userspace tools that deal with security policies and security associations are not great. They usually have terrible documentation and provide output that's not easy to parse (neither by humans nor tools). Ip xfrm is certainly no exception but it's a very powerful tool that can be used to help troubleshoot issues with the SPDB and SADB and even make changes to them in a reasonably straightforward fashion.

So what has IPsec ever done for us?

Encryption happens inside the kernel, so it's fast!

Using the right algorithms/settings it can be fairly secure

It's a standard, so there are good practices to use it securely

Very flexible, which is useful if you have:

- Hardware distributed across different datacenters/providers
- No real control over your network infrastructure

So what things is IPsec good for?

Once a full security association is established, in-kernel encryption is fast. This is very important when your internal network has high traffic levels, which is our case.

As complex as IPsec is, it is a standard, which means that a lot of people have looked at it and have established good practices that can be followed to secure it. With a good understanding of the protocols involved and choosing the right algorithms you can be reasonably confident that your traffic is going to be secured.

Part of the reasons for IPsec's complexity is also the flexibility it provides, which is actually a plus in our situation. This flexibility allows for IPsec to be used across two gateways or, in our case, in a full-mesh fashion with servers distributed across different locations and providers.

Hosted Graphite does IPsec!

Our migration: n2n -> IPsec

Big time constraints: n2n was unreliable and preventing us from scaling

We had trouble finding reports of people using IPsec in the same way*…

...So we had to improvise a bit.

After careful planning and testing we rolled it out to our production cluster...

* Notable exception: pagerduty's Doug Barth at Velocity 2015
http://conferences.oreilly.com/velocity/devops-web-performance-2015/public/schedule/detail/41454

So IPsec seemed to match our requirements reasonably well, particularly given that we weren't able to find anything else that did, so we decided to migrate our internal network to it.
Given that n2n's reliability record had been quite poor (and was only getting worse) and the lack of available IP space we needed to get familiar with IPsec and migrate in a fairly compressed timeframe.

When we began this project we expected to find a lot of reports of people using IPsec in a full-mesh setup as we planned on doing, but were surprised to find very little on the matter, up to the point where we started doubting if it might actually be a terrible idea. The only real exception for this was a talk given at Velocity 2015 describing a very similar system to what we were trying to implement at the time. We might have given up if we hadn't found that someone actually managed to solve the same problem with a very similar approach.

We managed to get a proof of concept up and running and we tested migrations and rollbacks in both testing and dedicated clusters receiving live traffic with no perceived impact, until we were reasonably confident to migrate our main production cluster to it.

It didn't go well, at all.


WORST


MIGRATION


EVER

WORST. MIGRATION. EVER

Migration attempt resulted in multi-day incident:

http://status.hostedgraphite.com/incidents/gw2v1rhm8p5g

Took two days to stabilize, a full week to resolve the incident.

Lots of issues not found during testing

So our first rollout was… not great. When rolling out on a bigger scale we ended up hitting a lot of issues none of our previous testing was able to detect.

The system got so unstable that it took us a couple of days to stabilize it and a week to recover all impact, we needed to stash away some data and replay it later.

Even though we had done a fair amount of testing, our testing wasn't done at quite the same scale as our production cluster sees

n2n -> IPsec migration aftermath

Back to drawing board, came up with another plan

Spent almost 3 months slowly rolling it out and fixing bugs:

- Also known as “the worst three months of my life”
- Big team effort, everybody pitched in

Still worth it, things are stable now and we’ve learned a lot

When the dust settled we found ourselves between a rock and a hard place, we still needed to migrate as quickly as possible but we couldn’t afford to keep breaking production until we managed to get it right.

We came up with a slower rollout process and started fixing things as they came along, it was a very slow process as some of the issues we found would only appear when dealing with hundreds of hosts and with production-level traffic, and it probably was the single most stressful project I’ve ever been involved with, and at times we even considered scrapping the whole thing and forgetting about it, but we did learn a few things about how not to run IPsec and, while still not perfect, it has greatly improved our internal network infrastructure.

Our stack: present day

Our IPsec stack: present day

Hundreds of hosts using ESP in transport mode (full-mesh)

Several clusters, isolated from each other

Using ipsec-tools with racoon as key management daemon

At this point we've fully migrated our internal network to IPsec, all internal communications are secured and different clusters are effectively isolated from each other as they belong to different IPsec groups.

In the linux world there's some degree of choice with regards to what key management daemon to use. They range from obsolete to awkward to configure. We ended up going with racoon, which allowed us to get started fairly quickly but which may not have been the smartest strategy long term for several reasons we'll explore further.

Our config: iptables

```
# Accept all IKE traffic, also allowing NAT Traversal (UDP 4500)
-A ufw-user-input -p udp --dport 500 -j ACCEPT
-A ufw-user-input -p udp --dport 4500 -j ACCEPT


# Allow  all ESP traffic, if it has a formed IPsec SA we trust it
-A ufw-user-input -p esp -j ACCEPT
```

Our IPsec-related iptables rules are pretty basic. We need to allow traffic on udp ports 500 and 4500 to allow IKE traffic through, and we also allow all esp-encapsulated traffic in, which means that after an IPsec SA has been established, two hosts will behave as if they're in the same network, with full network access to each other.

This simplifies our firewall rules considerably, and simpler rules are more secure rules.

Our config: Security Policies (`/etc/ipsec-tools.conf`)

Node1 = 1.2.3.4 Node2 = 5.6.7.8

On node1:

```
# require use of IPsec for all other traffic with node2
spdadd 1.2.3.4 5.6.7.8 any -P out ipsec esp/transport//require;
spdadd 5.6.7.8 1.2.3.4 any -P in ipsec esp/transport//require;
```

On node2:

```
# require use of IPsec for all other traffic with node1
spdadd 5.6.7.8 1.2.3.4 any -P out ipsec esp/transport//require;
spdadd 1.2.3.4 5.6.7.8 any -P in ipsec esp/transport//require;
```

In order for two hosts to use IPsec for securing their communications, you need to define the relevant security policies on both nodes.
A security policy matches a source and destination host and a protocol (in this case any protocol really) with an IPsec protocol/mode.

In this case we can see that ESP in transport mode is required for all communication between these two hosts. If we had used ‘use’ instead of ‘require’ IPsec would have been used if an SA is available, but the connection would have reverted to plain text if there were any issues establishing the SA. This is very useful while rolling out IPsec to a new cluster, as you can slowly set the policies to ‘require’ as you confirm everything works as intended.

Our config: Security Policies (`/etc/ipsec-tools.conf`)

What about management hosts?

Node1 = 1.2.3.4 PuppetMaster = 5.6.7.8

On node1:

```
# Only require IPsec for port 8140 on the puppet master
spdadd 1.2.3.4 5.6.7.8[8140] any -P out ipsec esp/transport//require;
spdadd 5.6.7.8[8140] 1.2.3.4 any -P in ipsec esp/transport//require;
```

Everything else will get dropped by the firewall

For management hosts such as puppet masters or syslog servers we can give them limited access to the other nodes, to help reduce the attack surface.
In this case only traffic to/from port 8140 in our puppet master will use IPsec, everything else will be in clear text (and thus dropped by the firewalls)

Our config: Security Policies (`/etc/ipsec-tools.conf`)

```
# Exclude ssh traffic:
spdadd 0.0.0.0/0[22] 0.0.0.0/0 tcp -P in prio def +100 none;
spdadd 0.0.0.0/0[22] 0.0.0.0/0 tcp -P out prio def +100 none;
spdadd 0.0.0.0/0 0.0.0.0/0[22] tcp -P in prio def +100 none;
spdadd 0.0.0.0/0 0.0.0.0/0[22] tcp -P out prio def +100 none;


# Exclude ICMP traffic (decouple ping and the like from IPsec):
spdadd 0.0.0.0/0 0.0.0.0/0 icmp -P out prio def +100 none;
spdadd 0.0.0.0/0 0.0.0.0/0 icmp -P in prio def +100 none;
```

You don’t want to be locked out of a server because IPsec is misbehaving, so you can explicitly set an exception for the SSH port.
It’s probably also a good idea to set another exception for all ICMP traffic, as it’s useful to be able to use ping to test the status of the network without wondering if it’s failing because of an underlying IPsec issue. There’s a similar issue with traceroute that we’ll expand on later.

Note how the exceptions have their priority set, to ensure they get processed before the other rules that might require IPsec usage between two hosts.

Our config: racoon (`/etc/racoon.conf`)

Phase 1:

```
remote anonymous {
    exchange_mode main;
    dpd_delay 0;
    lifetime time 24 hours;
    nat_traversal on;
    proposal {
        authentication_method pre_shared_key;
        dh_group modp3072;
        encryption_algorithm aes;
        hash_algorithm sha256;
    }
}
```

The racoon config is split in two parts, for phases 1 and 2.

Setting the remote to ‘anonymous’ allows us not to have to add new hosts to this config file every time hosts are added/removed to our cluster.

You can also see that we have disabled dead peer detection (more on that later), that our phase 1 relationships have a lifetime of 24 hours or that nat traversal is enabled. The proposal sets the algorithms and authentication mechanisms that will be negotiated with the other host.

Our config: racoon (`/etc/racoon.conf`)

Phase 2:

```
sainfo anonymous {
    pfs_group modp3072;
    encryption_algorithm aes;
    authentication_algorithm hmac_sha256;
    compression_algorithm deflate;

    lifetime time 8 hours;
}
```

And this is the configuration for phase 2.

These are the actual algorithms that will be used for the IPsec SAs. You can also see that PFS is enabled and that the lifetime of phase 2 relationships is of 8 hours in our case. The kernel will signal racoon before that time is up and it will get renegotiated.

10 DOS AND 500 DONT'S

Disclaimer:

(We don't really have 10 dos)

Don’t use ipsec-tools/racoon! (like we did)

Not actively maintained (Last release on early 2014)

Buggy

But the only thing that worked for us under time/resource constraints

LibreSwan seems like a strong alternative

In hindsight, it might be that our biggest mistake was to go with ipsec-tools/racoon. Racoon hasn’t actually seen a new release in the last two years and it hasn’t proven to be as stable as we initially hoped, as we’ve had to work around our share of bugs over the last year.

On the other hand, during our initial evaluation and rollout it’s the only alternative that we managed to get to work as intended with the time constraints that we had.

We will be evaluating libreswan in the near future hoping that we might be able to replace racoon with everything we’ve learned.

“The mystery of the disappearing SAs”

Some hosts unable to establish SAs on certain cases

Racoon would complain of SAs not existing (kernel would disagree):

```
ERROR: no policy found: id:281009.
```

racoon’s internal view of the SADB would get out of sync with the kernel’s

We suspect corruption in racoon’s internal state for the SADB

This is one example of a bug in racoon causing us terrible headaches.

We noticed that some hosts (usually after modifying their Security Policies) would be unable to establish SAs with other hosts until racoon was restarted.
Logs would complain about those SAs not existing in the SADB but dumps of the SADB itself would show that the SAs were in fact there.

Turns out that racoon keeps its own internal copy of the SADB which ends up being out of sync with the kernel’s under certain situations, like in some cases after quickly flushing the SPDB and SADB.
We even discovered that, even if left alone, racoon’s view of the SADB would differ from the kernel’s after a while on some hosts.

“The mystery of the disappearing SAs”

Restarting racoon fixes it, but that wipes out all your SAs!

Workaround: Force racoon to reload both SADB and config

```
killall -HUP racoon
```

Forcing periodic reloads prevents the issue from reoccurring ¯_(ツ)_/¯

Our first workaround was to also restart racoon when modifying the SPD, but the impact is just too big, as it forces you to restablish all your SAs.

Luckily, racoon can easily be made to refresh both its config and its internal view of the SADB easily.

Even doing this periodically can help prevent the issue from ever happening again with no perceived negative impact.

Don’t blindly force all traffic to go through IPsec

Account for everything that needs an exception:

- SSH, ICMP, etc

You’ll need to be able to answer these two questions:

- “Is the network broken?”
- “Is IPsec broken?”

It’s important when crafting our security policies to try to identify all traffic that we might *not* want to go through IPsec.
We’ve already seen that SSH is one example, as we don’t want to lose access to our servers if there’s an IPsec problem.
Making an exception for ICMP gives you an easy way to check if connectivity between two hosts is affected by a network issue or an IPsec issue.

“Yo dawg, I heard you like encrypted traffic…”

If migrating from an existing VPN, make sure to exclude it from IPsec traffic

During our initial rollout our SPs forced our n2n traffic through IPsec…

… Which still wasn’t working reliably enough…

… Effectively killing our whole internal network

One other example of traffic that should be exempt from going through IPsec might be other VPN traffic.
In our case, while we were transitioning from n2n to IPsec I neglected making an exception for n2n traffic, which meant that the kernel would try to wrap n2n traffic through IPsec, which was still not working as reliably as intended, causing impact on our internal network… and our backup internal network.

Don’t just enable DPD… without testing

What’s DPD?

- DPD: Dead Peer Detection (RFC3706)
- Liveness checks on Phase 1 relationships
- If no response to R-U-THERE clears phase 1 and 2 relationships…

Sounds useful but test it in your environment first:

- racoon implementation is buggy!

One of the thousand chioces you need to make when configuring IPsec is whether to enable DPD or not.
DPD enables liveness checks for phase 1 relationships, sending R-U-THERE messages to the other node if it hasn’t seen traffic in a while. After several failures to respond it will assume the other side is dead and clear both phase 1 and 2 relationships.

“The trouble with DPDs”

In our case, enabling DPD results in 100s of SAs between two hosts:

- Every failed DPD check resulting in extra SAs

Combination of factors:

- Unreliable network
- Bugs in racoon

We ended up giving up on DPD

For us, we tried with several different values for delays and number of attempts with the same result, the number of phase 2 SAs would keep increasing and increasing until we would have hundreds of relationships between two hosts.

We know our network is not entirely reliable but that doesn’t explain this, as the expected behaviour is to *clear* the existing relationships if they’re considered dead. We ended up giving up on DPD as the headaches were outweighing the benefits.

Don’t just disable DPD either

DPD can be legitimately useful

Example: What happens when rebooting a host?

Other nodes might not realise their SAs are no longer valid!

But DPD does serve a purpose!

Being able to detect network connectivity issues and refresh the SAs is useful, as is being able to detect when the other side of a connection has been rebooted.
If one of the nodes in a connection is rebooted the other side might not realise that the SAs are no longer valid, causing connectivity issues.

DPD: Rebooting hosts

bender's SAD:

```
5.6.7.8 -> 1.2.3.4 (spi: 0x01)

1.2.3.4 -> 5.6.7.8 (spi: 0x02)
```

flexo's SAD:

```
5.6.7.8 -> 1.2.3.4 (spi: 0x01)

1.2.3.4 -> 5.6.7.8 (spi: 0x02)
```

These are two happy hosts right now…

bender -> flexo (using spi `0x02`) traffic is received by flexo

flexo -> bender (using spi `0x01`) traffic is received by bender!

… But let's say we reboot bender!

Let's look at an example. Say we have two hosts with mature SAs to each other. They can talk to each other and everything works.

At this point we reboot bender!

DPD: Rebooting hosts

bender's SAD:

~~5.6.7.8 -> 1.2.3.4 (spi: 0x02)~~

~~1.2.3.4 -> 5.6.7.8 (spi: 0x01)~~

flexo's SAD:

```
5.6.7.8 -> 1.2.3.4 (spi: 0x02)
```

```
1.2.3.4 -> 5.6.7.8 (spi: 0x01)
```

bender's SADB is now empty

flexo->bender traffic (using spi 0x02) will be broken until:

- bender->flexo traffic forces establishment of new SAs
- The SAs on flexo's side expire

When bender comes back online its SADB will be empty, so as far as bender is concerned it has no active SAs with flexo.
flexo still thinks it has valid SAs with bender, so if we try to initiate a connection from flexo to bender it will attempt to use that SA (spi 0x02) and the connection will be dropped by bender, as it doesn't know anything about that SA with spi 0x02

If we initiate a connection from bender to flexo instead everything will work, as bender will renegotiate a new phase1 SA before attempting to connect, so the new SAs will be used.
Also, if you wait for long enough those SAs will expire and everything will work again.

None of those are actual solutions, obviously.

DPD: Just roll your own

Our solution: Implement our own phase 2 liveness check

Check a known port for every host we have a mature SA with:

- Clear the SAs if ${max_tries} timeouts

Bonus points: Also check a port that won't use IPsec to compare

Given that DPD doesn't check phase 2 relationships the solution is simple: implement our own phase 2 liveness check.

We go over all the hosts we have mature SAs with and attempt to connect to an unused port.
Our firewall rules will drop all non-IPsec traffic so a successful test means getting a connection refused from the other side.
After several failures we can log the error and clear the SA.

For bonus points implementing the same check on an IPsec-exempt port will tell you if the problem is IPsec-related or a more general network issue.

Do instrument all the things!

You'll ask yourself "is this the network or IPsec?" a lot

So better have lots of data!

Built racoon to emit timing info on logs (build with `--enable-stats`)

Diamond collector gathers and send metrics from:

- racoon logs
- SADB

You'll never be sure if a connectivity issue is a network vs IPsec issue unless you have data.

One of our biggest wins when rolling out IPsec to production was to build racoon with stats enabled and got a diamond collector to emit metrics.
We also wrote a custom collector that dumps the current state of the SADB and emits several key metrics.

Instrumenting the racoon logs

As an example of metrics you can get from racoon, we have timing data for forming both phase 1 and phase 2 relationships.

Phase 2 relationships are more common as there are two per node pair and their lifetime is much shorter than for phase 1 relationships.

Instrumenting the SADB

Dumping state from the SADB can also be very useful. By keeping an eye on “larval” relationships we can know when a host is struggling to establish SAs to other hosts. We can also correlate performance issues with the existing SAs being expired and new ones being renegotiated.

These are two examples of healthy graphs where relationships close to the end of their lifetime get replaced without much fuss.

Do instrument all the things!

Kernel metrics also useful (if available!)

You want your kernel compiled with `CONFIG_XFRM_STATISTICS`

```
$ cat /proc/net/xfrm_stat
XfrmInError                     0
XfrmInBufferError               0
…
```

(Very) brief descriptions: https://www.kernel.org/doc/Documentation/networking/xfrm_proc.txt

Kernel metrics are really really useful, if you have access to them.

Some distros (ubuntu I'm looking at you) shipped kernels not compiled with CONFIG_XFRM_STATISTICS, but latest kernels seem to be ok, your mileage may vary though.

There's not a lot of documentation on what those values actually *mean*, and in some cases even some source digging might be required, but when you reach for them you'll probably get what you want (provided you know where to look)

Instrumenting kernel xfrm stats: XfrmInNoStates

This is an example of one of the xfrm metrics being useful during an incident. XfrmInNoStates is incremented when we’re receiving data for a SA that we don’t have (or with wrong spi, etc)

We can see a spike on Wednesday that’s correlated with a bunch of reported internal network issues. Having access to this metric was vital in order to quickly figure out what was going on. Without access to this metric we would have continued guessing for ages.

“The case of the sad halfling”

bender’s SAD:

```
5.6.7.8 -> 1.2.3.4 (spi: 0x02)

1.2.3.4 -> 5.6.7.8 (spi: 0x01)
```

flexo’s SAD:

```
5.6.7.8 -> 1.2.3.4 (spi: 0x02)

1.2.3.4 -> 5.6.7.8 (spi: 0x01)
```

These are two happy hosts right now...

… But let’s say one SA “disappears” during a brief netsplit:

```
bender$ echo “deleteall 5.6.7.8 1.2.3.4 esp ; ” | setkey -c

# The 5.6.7.8 1.2.3.4 association gets removed from bender
```

The previous incident was caused by a variety of networking issues on our hosting provider but we had never seen our IPsec stack reacting in that way to network issues before.
One of our SREs put together a repro case which I think highlights very well the need for both metrics and having your own checks.

Say we have two hosts, same as before, with full connectivity.
At some point the flexo->bender SA gets removed from bender. If that happens when both hosts have full network connectivity bender should notify flexo and both would remove it, that would be the end of it. But what happens if it gets removed during a brief netsplit?

“The case of the sad halfling”

bender’s SAD:

~~5.6.7.8 -> 1.2.3.4 (spi: 0x02)~~

1.2.3.4 -> 5.6.7.8 (spi: 0x01)

flexo’s SAD:

5.6.7.8 -> 1.2.3.4 (spi: 0x02)

1.2.3.4 -> 5.6.7.8 (spi: 0x01)

Any communication attempt will fail!

bender -> flexo (using spi 0x01) traffic is received by flexo

flexo -> bender (using spi 0x02) traffic is ignored by bender!

The SAs are not symmetrical now and all communication attempts will fail.

flexo will always use SA 0x02 to send data to bender, but bender doesn’t recognise that SA anymore, so it will be dropped. Even if we initiate communication from bender, the replies will never make it to bender.

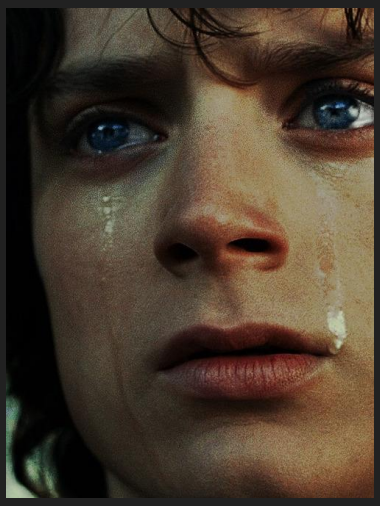
That’s the effect we noticed during that incident, that some hosts would only have 1 mature SA between two hosts, when two were expected.

“The case of the sad halfling”

We built a custom daemon for detecting it

Highlights need for:

- Phase 2 liveness checks
- Metrics for everything!



Once we knew what to look for, we wrote a daemon to detect when this was happening and clear the affected SAs. This ended up evolving into a more general phase 2 liveness check that we discussed previously.

Really important to stress the need for liveness checks and having the right metrics in place.

Don’t flush/restart on changes

Never restart racoon!

A racoon restart will flush all phase 1 and 2 SAs:

- Negotiating ~1000 SAs at once is no fun

To flush an individual SA: `ip xfrm state delete`

To reload config changes: `killall -HUP racoon`

This was a big problem for us initially, actually figuring out how to make changes to cluster without triggering thousands of reassociations that had a noticeable impact. This might sound stupid, but you don’t want to restart racoon, like ever. Restarting racoon implies clearing both your phase 1 and phase 2 relationships, and having hosts renegotiate 1000 SAs each at the same time is no fun at all.

Thankfully racoon will happily reload its configuration without a restart, and the same applies to pre-shared keys.

Don’t flush/restart on changes

When adding/removing hosts, do not flush both SPD and SAD!

Instead, consider just flushing your SPD and letting unwanted SAs to expire

SAs not reflected in the SPD will never get used

Can flush that SA individually if feeling paranoid

Can just include `spdflush;` in your `ipsec-tools.conf` and reload with:

```
setkey -f /etc/ipsec-tools.conf
```

In modern environments, adding and removing hosts is very common, so we can’t really afford to flush all our SAs every time there’s a change to our host list. By ensuring your SADB is left unchanged you’re minimizing any potential disruption to your existing connections.

Say you’re retiring a host, and you’d like to remove it from all security policies. If you remove it from the SPD it won’t be able to initiate/receive any IPsec traffic with any other host, even if it still has an active SA. You can just wait for the SA to expire at its own time.

This means you can get away with never flushing your SADB, which is really valuable.

You don’t have the same tools available

tcpdump will just show ESP traffic, not its content:

```
15:47:51.511135 IP 1.2.3.4 > 5.6.7.8: ESP(spi=0x00fb0c52,seq=0x1afa), length 84

15:47:51.511295 IP 5.6.7.8 > 1.2.3.4: ESP(spi=0x095e5523,seq=0x173a), length 84
```

Traffic can be decrypted with wireshark/tshark if you dump the keys first

When running your internal network on IPsec some tools will behave differently or not be available. One example of this is tcpdump.

Tcpdump will just show you there’s esp-encapsulated traffic between two hosts, but won’t be able to show you the contents of such traffic.

If you have a traffic capture and dump the contents of your SADB you can configure wireshark/tshark to decrypt the traffic. We’ve had mixed results getting tshark to capture/decrypt traffic on the fly, never enough to serve as a drop-in tcpdump replacement.

You don’t have the same tools available

Can use tcpdump with netfilter logging framework:

```
$ iptables -t mangle -I PREROUTING -m policy --pol ipsec --dir in -j NFLOG --nflog-group 5

$ iptables -t mangle -I POSTROUTING -m policy --pol ipsec --dir out -j NFLOG --nflog-group 5

$ tcpdump -i nflog:5
```

Doesn’t allow most filters

Might need to increase the buffer size

We’re currently working around the issue using the netfilter logging framework. We wrote a script that logs all IPsec traffic and then we can get tcpdump to capture that traffic. It’s still not perfect as, for example, the nflog interface has really limited (nonexistent really) filtering capabilities, but it’s useful. We might be able to achieve better filtering by playing with the iptables rules.

You don’t have the same tools available

Traceroute will attempt to use udp by default:

```
$ traceroute that.other.host
traceroute to that.other.host (5.6.7.8), 30 hops max, 60 byte packets
 1  * * *
 2  * * *
 3  * * *
 4  that.other.host (5.6.7.8)  0.351 ms  0.295 ms  0.297 ms
```

You can force it to use ICMP with `traceroute -I`

Tools like traceroute and mtr will break by default with IPsec, which can be a bit surprising. In the case of traceroute the solution is as simple as forcing it to use ICMP (provided we’re exempting ICMP traffic from going through IPsec).

Do use certs for auth, or don't use a weak PSK

PSK is great for getting started if you don't have PKI in place (we didn't)

But please:

- Use a strong PSK (if you must use PSK)
- Enable PFS (Perfect Forward Secrecy)
- Do not use aggressive mode for phase 1

Not following all that makes the NSA happy!

If you have a public key infrastructure, you should really be using certs.
If you don't, Pre-shared keys are a good starting point but please follow this at the very least.
The NSA has been known to exploit IPsec network with weak PSKs that don't use PFS, and/or use aggressive mode for phase 1.

Enabling PFS ensures that new encryption keys will be generated independently of previous keys, so if a previous key is compromised your communication is still secure. Even if an attacker were to break your IKE SA and attacker can not compromised your IPsec SAs.

Don’t trust the (kernel) defaults!

Careful with `net.ipv4.xfrm4_gc_thresh`

Associations might be garbage collected before they can succeed!

If 3.6 > `$(uname -r)` < 3.13:

- Default `(1024)` might be too low
- GC will cause performance issues
- Can refuse new allocations if you hit `(1024 * 2)` dst entries

Careful with your kernel settings!

The settings for the xfrm garbage collector have been jumping up and down for a while, so make sure yours has a sane value. If the value is too low the gc might even kick in while you’re trying to form a relationship, and kill it with fire. It can also refuse new allocations when you hit the (reasonably small) limit.

Don’t trust the defaults!

Beware of IPsec overhead and MTU/MSS:

A 1450 bytes IP packet becomes:

- 1516 bytes in Transport mode
- 1532 bytes in Tunnel mode

(More if enabling NAT Traversal!)

Path MTU Discovery should help, but test it first!

This may or may not be an issue for you, but you should definitely test it.
IPsec adds overhead so it can push you over the MTU if you’re not careful. Assuming an MTU of 1500 it can easily push you over the edge (particularly on tunnel mode due to larger overhead)

Thanks!

Questions?

Hated it? Want to say hi?

fran@hostedgraphite.com

@hostedgraphite

hostedgraphite.com/jobs