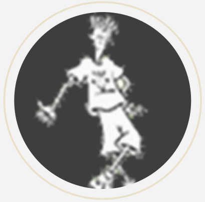

Masterzen

You'll find here my current and past thoughts about my work and hobbies, including but not limited to system administration, programming, photography ,boardgames or mechanical keyboards.

Somewhere

Puppet SSL explained

14 minute read

The puppet-users or #puppet freenode irc channel is full of questions or people struggling about the puppet SSL PKI. To my despair, there are also people wanting to completely get rid of any security.

While I don't advocate the *live happy, live without security* motto of some puppet users (and I really think a corporate firewall is only one layer of defense among many, not the ultimate one), I hope this blog post will help them shoot themselves in their foot :)

I really think SSL or the X509 PKI is simple once you grasped its underlying concept. If you want to know more about SSL, I really think everybody should read Eric Rescola's excellent “SSL and TLS: Designing and Building Secure Systems”.

I myself had to deal with SSL internals and X509 PKI while I implemented a java secured network protocol in a previous life, including a cryptographic library.

Purpose of Puppet SSL PKI

The current puppet security layer has 3 aims:

1. authenticate any node to the master (so that no rogue node can get a catalog from your master)
2. authenticate the master on any node (so that your nodes are not tricked into getting a catalog from a rogue master).
3. prevent communication eavesdropping between master and nodes (so that no rogue users can grab configuration secrets by listening to your traffic, which is useful in the cloud)

A notion of PKI

PKI means: Public Key Infrastructure. But whats this?

A PKI is a framework of computer security that allows authentication of individual components based on public key cryptography. The most known system is the x509 one which is used to protect our current web.

A public key cryptographic system works like this:

- every components of the system has a secret key (known as the *private key*) and a *public key* (this one can be shared with other participant of the system). The public and private keys are usually bound by a cryptographic algorithm.
- authentication of any component is done with a simple process: a component signs a message with its own private key. The receiver can authenticate the message (ie know the message comes from the original component) by validating the signature. To do this, only the public key is needed.

There are different public/private key pair cryptosystem, the most known ones are RSA, DSA or those based on Elliptic Curve cryptography.

Usually it is not good that all participants of the system must know each other to communicate. So most of the current PKI system use a hierarchical validation system, where all the participant in the system must only know one of the parent in the hierarchy to be able to validate each others.

X509 PKI

X509 is an ITU-T standard of a PKI. It is the base of the SSL protocol authentication that puppet use. This standard specifies certificates, certificate revocation list, authority and so on…

A given X509 certificate contains several information like those:

- Serial number (which is unique for a given CA)
- Issuer (who created this certificate, in puppet this is the CA)
- Subject (who this certificate represents, in puppet this is the node certname or fqdn)
- Validity (valid from, expiration date)
- Public key (and what kind of public key algorithm has been used)
- Various extensions (usually what this certificate can be used for,…)

You can check RFC1422 for more details.

The certificate is usually the DER encoding of the ASN.1 representation of those informations, and is usually stored as PEM for consumption.

A given X509 certificate is signed by what we call a Certificate Authority (CA for short). A CA is an infrastructure that can sign new certificates. Anyone sharing the public key of the CA can validate that a given certificate has been validated by the CA.

Usually X509 certificate embeds a RSA public key with an exponent of 0x100001 (see below). Along with a certificate, you need a private key (usually also PEM-encoded).

So basically the X509 system works with the following principle: CA are using their own private keys to sign components certificates, it is the CA role to sign only trusted component certificates. The trust is usually established out-of-bound of the signing request.

Then every component in the system knows the CA certificate (ie public key). If one component gets a message from another component, it checks the attached message signature with the CA certificate. If that validates, then the component is authenticated. Of course the component should also check the certificate validity, if the certificate has been revoked (from OCSP or a given CRL), and finally that the certificate subject matches who the component pretends to be (usually this is an hostname validation against some part of the certificate *Subject*)

RSA system

Most of X509 certificate are based on the RSA cryptosystem, so let’s see what it is.

The RSA cryptosystem is a public key pair system that works like this:

Key Generation

To generate a RSA key, we chose two prime number p and q.

We compute $n=pq$. We call n the modulus.

We compute $\varphi(pq) = (p - 1)(q - 1)$.

We chose e so that e>1 and e<$\varphi(pq)$ (e and $\varphi(pq)$ must be coprime). *e* is called the exponent. It usually is 0x10001 because it greatly simplifies the computations later (and you know what I mean if you already implemented this :)).

Finally we compute *d=e^-1 mod((p-1)(q-1))*. This will be our secret key. Note that it is not possible to get d from only e (and since p and q are never kept after the computation this works).

In the end:

- *e* and *n* form the public key
- *d* is our private key

Encryption

So the usual actors when describing cryptosystems are Alice and Bob. Let's use them.

Alice wants to send a message *M* to Bob. Alice knows Bob's public key *(e,n)*. She transform *M* in a number < *n _(this is called padding) that we'll call _m*, then she computes: _c = m^e . mod(n) _

Decryption

When Bob wants to decrypt the message, he computes with his private key *d*: *m = c^d . mod(n)*

Signing message

Now if Alice wants to sign a message to Bob. She first computes a hash of her message called *H*, then she computes: _s = H^(d mod n). _So she used her own private key. She sends both the message and the signature.

Bob, then gets the message computes _H _and computes _h' = H^(e mod n) _with Alice's public key. If _h' = h, _then only Alice could have sent it.

Security

What makes this scheme work is the fundamental that finding p and q from n is a hard problem (understand for big values of n, it would take far longer than the validity of the message). This operation is called factorization. Current certificate are numbers containing 2048 bits, which roughly makes a 617 digits number to factor.

Want to know more?

Then there are a couple of books really worth reading:

- Applied Cryptography - Bruce Schneier
- Handbook of Applied Cryptography - Alfred Menezes, Paul van Oorschot, Scott Vanstone

How does this fit in SSL?

So SSL (which BTW means Secure Socket Layer) and now TLS (SSL successor) is a protocol that aims to provide security of communications between two peers. It is above the transport protocol (usually TCP/IP) in the OSI model. It does this by using symmetric encryption and message authentication code (MAC for short). The standard is (now) described in RFC5246.

It works by first performing an handshake between peers. Then all the remaining communications are encrypted and tamperproof.

This handshake contains several phases (some are optional):

1. Client and server finds the best encryption scheme and MAC from the common list supported by both the server and the clients (in fact the server choses).
2. The server then sends its certificate and any intermediate CA that the client might need
3. The server may ask for the client certificate. The client may send its certificate.
4. Both peers may validate those certificates (against a common CA, from the CRL, etc...)
5. They then generate the session keys. The client generates a random number, encrypts it with the server public key. Only the server can decrypt it. From this random number, both peers generate the symmetric key that will be used for encryption and decryption.

6. The client may send a signed message of the previous handshake message. This way the server can verify the client knows his private key (this is the client validation). This phase is optional.

After that, each message is encrypted with the generated session keys using a symmetric cipher, and validated with an agreed on MAC. Usual symmetric ciphers range from RC4 to AES. A symmetric cipher is used because those are usually way faster than any asymmetric systems.

Application to Puppet

Puppet defines it's own Certificate Authority that is usually running on the master (it is possible to run a CA only server, for instance if you have more than one master).

This CA can be used to:

- generate new certificate for a given client out-of-bound
- sign a new node that just sent his Certificate Signing Request
- revoke any signed certificate
- display certificate fingerprints

What is important to understand is the following:

- Every node knows the CA certificate. *This allows to check the validity of the master from a node*

- *The master doesn't need the node certificate*, since it's sent by the client when connecting. It just need to make sure the client knows the private key and this certificate has been signed by the master CA.

It is also important to understand that when your master is running behind an Apache proxy (for Passenger setups) or Nginx proxy (ie some mongrel setups):

- The proxy is the SSL endpoint. It does all the validation and authentication of the node.
- Traffic between the proxy and the master happens in clear
- The master knows the client has been authenticated because the proxy adds an HTTP header that says so (usually _X-Client-Verify _for Apache/Passenger).

When running with webrick, webrick runs inside the puppetmaster process and does all this internally. Webrick tells the master internally if the node is authenticated or not.

When the master starts for the 1st time, it generates its own CA certificate and private key, initializes the CRL and generates a special certificate which I will call the *server certificate*. This certificate will be the one used in the SSL/TLS communication as the server certificate that is later sent to the client. This certificate subject will be the current master FQDN. If your master is also a client of itself (ie it runs a puppet agent), I recommend using this certificate as the client certificate.

The more important thing is that this server certificate advertises the following extension:

```
X509v3 Subject Alternative Name:
                DNS:puppet, DNS:$fqdn, DNS:puppet.$domain
```

What this means is that this certificate will validate if the connection endpoint using it has any name matching puppet, the current fqdn or puppet in the current domain.

By default a client tries to connect to the "*puppet*" host (this can be changed with –server which I don't recommend and is usually the source of most SSL trouble).

If your DNS system is well behaving, the client will connect to *puppet.$domain*. If your DNS contains a CNAME for puppet to your *real master fqdn*, then when the client will validate the server certificate it will succeed because it will compare "puppet" to one of those DNS: entries in the aforementioned certificate extension. BTW, if you need to change this list, you can use the –certdnsname option (note: this can be done afterward, but requires to re-generate the server certificate).

The whole client process is the following:

1. if the client runs for the 1st time, it generates a Certificate Signing Request and a private key. The former is an x509 certificate that is self-signed.
2. the client connects to the master (at this time the client is not authenticated) and sends its CSR, it will also receives the CA certificate and the CRL in return.
3. the master stores locally the CSR
4. the administrator checks the CSR and can eventually sign it (this process can be automated with autosigning). I strongly suggest verifying certificate fingerprint at this stage.

5. the client is then waiting for his signed certificate, which the master ultimately sends
6. All next communications will use this client certificate. Both the master and client will authenticate each others by virtue of sharing the same CA.

Tips and Tricks

Troubleshooting SSL

Certificate content

First you can check any certificate content with this:

```
openssl x509 -text -in /var/lib/puppet/ssl/certs/puppet.pem

Certificate:    Data:
        Version: 3 (0x2)
        Serial Number: 2 (0x2)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: CN=Puppet CA: master.domain.com
        Validity
            Not Before: Nov 13 14:29:23 2010 GMT
            Not After : Nov 12 14:29:23 2015 GMT
        Subject: CN=server.domain.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:be:11:7d:0e:32:4d:c4:da:40:7d:7a:17:30:2c:
                    00:c4:c5:a8:c7:91:31:21:71:50:ef:07:77:79:1a:
                    07:d6:57:d4:4d:e0:01:b3:78:73:ec:84:dd:71:30:
                    62:cd:e5:26:fd:54:46:da:e3:3b:be:3b:05:9a:87:
                    44:9a:5e:b4:41:b7:15:de:20:1d:9d:26:50:44:bc:
```

```
                    e6:64:67:d1:93:ee:3f:20:a6:86:0e:11:5c:de:b1:
                    da:e5:fb:b5:f1:e1:e9:2e:14:39:47:f2:b8:a4:40:
                    84:89:18:86:5a:df:3b:68:a4:64:7f:a9:99:93:60:
                    29:e8:fe:d5:a3:e0:6e:ba:4b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            Netscape Comment:
                Puppet Ruby/OpenSSL Generated Certificate
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                F4:FA:5A:03:EF:D5:0C:C3:B6:A0:35:47:D1:49:98:74:D4:09:B4:A9
            X509v3 Key Usage:
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication, E-mail Protection
            X509v3 Subject Alternative Name:
                DNS:puppet, DNS:puppet.domain.com
    Signature Algorithm: sha1WithRSAEncryption
        70:e3:7c:04:c4:e1:66:07:db:5c:58:d9:64:bb:0a:e7:55:4c:
        93:9d:61:0a:2a:a6:3f:de:aa:98:a9:e5:40:45:40:87:62:78:
        d3:af:a7:01:a7:b9:ca:ee:b2:44:ff:02:be:8b:54:aa:65:45:
        0b:94:2a:56:fa:1d:67:fe:cd:52:09:29:89:bc:2f:4f:6b:30:
        cb:de:6a:01:35:43:74:1e:d6:14:2e:f0:43:ac:38:e9:7c:ec:
        2c:e6:b8:50:8c:15:07:2f:72:35:82:7f:ad:9c:3a:4f:a7:5c:
        d6:e8:87:f9:19:20:1f:8f:2e:2e:28:4c:9f:ea:d7:26:5e:c5:
        18:57
```

puppet-ssl.sh hosted with ❤ by **GitHub** **view raw**

Simulate a SSL connection

You can know more information about a SSL error by simulating a client connection. Log in the trouble node and:

```
# this simulates how a puppet agent will connect
openssl s_client -host puppet -port 8140 -cert /path/to/ssl/certs/node.domain.com.pem -key /path/to/

```

```
# outputs:

CONNECTED(00000004)
depth=1 /CN=Puppet CA: master.domain.com
verify return:1
depth=0 /CN=macbook.local
verify return:1
---
Certificate chain
 0 s:/CN=macbook.local
   i:/CN=Puppet CA: master.domain.com
 1 s:/CN=Puppet CA: master.domain.com
   i:/CN=Puppet CA: master.domain.com
---
Server certificate
-----BEGIN CERTIFICATE-----
MIICgjCCAeugAwIBAgIBAjANBgkqhkiG9w0BAQUFADAjMSEwHwYDVQQDDBhQdXBw
ZXQgQ0E6IG1hY2Jvb2subG9jYWwwHhcNMTAxMTEzMTQyOTIzWhcNMTUxMTEyMTQy
OTIzWjAYMRYwFAYDVQQDDA1tYWNib29rLmxvY2FsMIGfMA0GCSqGSIb3DQEBAQUA
A4GNADCBiQKBgQC+EX0OMk3E2kB9ehcwLADExajHkTEhcVDvB3d5GgfWV9RN4AGz
eHPshN1xMGLN5Sb9VEba4zu+OwWah0SaXrRBtxXeIB2dJlBEvOZkZ9GT7j8gpoYO
EVzesdrl+7Xx4ekuFDlH8rikQISJGIZa3ztopGR/qZmTYCno/tWj4G66SwIDAQAB
o4HQMIHNMDgGCWCGSAGG+EIBDQQrFilQdXBwZXQgUnVieS9PcGVuU1NMIEdlbmVy
YXRlZCBDZXJ0aWZpY2F0ZTAMBgNVHRMBAf8EAjAAMB0GA1UdDgQWBBT0+loD79UM
w7agNUfRSZh01Am0qTALBgNVHQ8EBAMCBaAwJwYDVR0lBCAwHgYIKwYBBQUHAwEG
CCsGAQUFBwMCBggrBgEFBQcDBDAuBgNVHREEJzAlggZwdXBwZXSCDW1hY2Jvb2su
bG9jYWyCDHB1cHBldC5sb2NhbDANBgkqhkiG9w0BAQUFAAOBgQBw43wExOFmB9tc
WNlkuwrnVUyTnWEKKqY/3qqYqeVARUCHYnjTr6cBp7nK7rJE/wK+i1SqZUULlCpW
+h1n/s1SCSmJvC9PazDL3moBNUN0HtYULvBDrDjpfOws5rhQjBUHL3I1gn+tnDpP
p1zW6If5GSAfjy4uKEyf6tcmXsUYVw==
-----END CERTIFICATE-----
subject=/CN=puppet.domain.com
issuer=/CN=Puppet CA: master.domain.com
---
No client certificate CA names sent
---
SSL handshake has read 1794 bytes and written 1656 bytes
---
```

```
New, TLSv1/SSLv3, Cipher is DHE-RSA-AES256-SHA
Server public key is 1024 bit
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : DHE-RSA-AES256-SHA
    Session-ID: DB29414CCB1E094675238999C8C00AF3173F441030C44A67D773648E83D76F75
    Session-ID-ctx:
    Master-Key: 92430ADC9E52BA22023D5E37DED7D9A274B9E5E461CB46C47F1E9B14BE1956B7615FADC2319D9DA0917
    Key-Arg   : None
    Start Time: 1289747911
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

Check the last line of the report, it should say "Verify return code: 0 (ok)" if both the server and client authenticated each others. Check also the various information bits to see what certificate were sent. In case of error, you can learn about the failure by looking that the verification error message.

ssldump

Using ssldump or wireshark you can also learn more about ssl issues. For this to work, it is usually needed to force the cipher to use a simple cipher like RC4 (and also ssldump needs to know the private keys if you want it to decrypt the application data).

Some known issues

Also, in case of SSL troubles make sure your master isn't using a different $ssldir than what you are thinking. If that happens, it's possible your master is using a different dir and has regenerated its CA. If that happens no one node can connect to it anymore.

This can happen if you upgrade a master from gem when it was installed first with a package (or the reverse).

If you regenerate a host, but forgot to remove its cert from the CA (with puppetca –clean), the master will refuse to sign it. If for any reason you need to fully re-install a given node without changing its fqdn, either use the previous certificate or clean this node certificate (which will automatically revoke the certificate for your own security).

Looking to the CRL content:

```
# it is possible to get the content of the CRL:
openssl crl -text -in /var/lib/puppet/ssl/ca/ca_crl.pem

Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: /CN=Puppet CA: master.domain.com
        Last Update: Nov 14 15:47:42 2010 GMT
        Next Update: Nov 13 15:47:42 2015 GMT
        CRL extensions:
            X509v3 CRL Number:
                1
Revoked Certificates:
    Serial Number: 03
        Revocation Date: Nov 14 15:47:42 2010 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    Signature Algorithm: sha1WithRSAEncryption
        a2:cb:cf:d6:95:34:5d:7e:aa:95:cf:cd:7f:ea:1a:da:b0:f4:
        15:1f:df:03:28:64:b7:e0:a9:2d:53:df:b7:25:05:64:3e:15:
        08:2a:02:6d:42:7f:ad:37:f1:8f:72:66:f5:ed:f0:0b:59:d2:
        9f:16:77:18:eb:dc:dd:2e:f0:c4:ea:80:51:cf:35:43:ed:cd:
        7d:64:c0:43:dc:85:13:0f:5f:e2:88:78:a9:fc:bf:c3:a5:c6:
        e2:0e:8e:9d:95:1e:19:63:03:bb:26:89:9c:52:78:d6:a0:79:
        82:1d:2c:44:15:7d:75:42:52:4e:6a:a8:e5:d7:40:c5:b8:4a:
              24:d2
```

crl.sh hosted with ❤ by GitHub view raw

Notice how the certificate serial number 3 has been revoked.

Fingerprinting

Since puppet 2.6.0, it is possible to fingerprint certificates. If you manually sign your node, it is important to make sure you are signing the correct node and not a rogue system trying to pretend it is some genuine node. To do this you can get the certificate fingerprint of a node by running puppet agent –fingerprint, and when listing on the master the various CSR, you can make sure both fingerprint match.

```
# on the node
puppet agent --test --fingerprint
notice: 14:45:FD:59:F2:CC:83:62:4C:4A:D2:2A:37:4F:12:96

# on the master
puppetca --list node.domain.com --fingerprint
node.domain.com 14:45:FD:59:F2:CC:83:62:4C:4A:D2:2A:37:4F:12:96
```

fingerprint.sh hosted with ❤ by GitHub view raw

Dirty Trick

Earlier I was saying that when running with a reverse proxy in front of Puppet, this one is the SSL endpoint and it propagates the authentication status to Puppet.

I strongly don't recommend implementing the following. This will compromise your setup security.

This can be used to severely remove Puppet security for instance you can:

- make so that every nodes are authenticated for the server by always returning the correct header
- make so that nodes are authenticated based on their IP addresses or fqdn

You can even combine this with a mono-certificate deployment. The idea is that every node share the same certificate. This can be useful when you need to provision tons of short-lived nodes. Just generate on your master a certificate:

```
# Generate a certificate and private key to be used for a node
puppetca --generate node.domain.com

notice: node.domain.com has a waiting certificate requestnotice: Signed certificate request for node.
notice: Removing file Puppet::SSL::CertificateRequest node.domain.com at '/tmp/master/ssl/ca/requests
notice: Removing file Puppet::SSL::CertificateRequest node.domain.com at '/tmp/master/ssl/certificate
```

You can then use those generated certificate (which will end up in /var/lib/puppet/ssl/certs and /var/lib/puppet/private_keys) in a pre-canned $ssldir, provided you rename it to the local fqdn (or symlink it). Since this certificate is already signed by the CA, it is valid. The only remaining issue is that the master will serve the catalog of this certificate certname. I proposed a patch to fix this, this patch will be part of 2.6.3. In this case the master will serve the catalog of the given connecting node and not the connecting certname. Of course you need a relaxed auth.conf:

```
...
path ~ ^/catalog/([^/]+)$
method find
allow $1
allow node.domain.com
...
```

Caveat: I didn’t try, but it should work. YMMV :)

Of course if you follow this and shoot yourself in the foot, I can’t be held responsible for any reasons, you are warned. Think twice and maybe thrice before implementing this.

Multiple CA or reusing an existing CA

This goes beyond the object of this blog post, and I must admit I never tried this. Please refer to: Managing Multiple Certificate Authorities and Puppet Scalability

Conclusion

If there is one: **security is necessary when dealing with configuration management**. We don’t want any node to trust rogue masters, we don’t want masters to distribute sensitive configuration data to rogue nodes. We even don’t want a rogue user sharing the same network to read the configuration traffic. Now that you fully understand SSL, and the X509 PKI, I’m sure you’ll be able to design some clever attacks against a Puppet setup :)

Previous Next

COMMENTS