Hardening Web Server

Muhammad Moinur Rahman<moin@bofh.im>
Philip Paeps<philip@trouble.is>

Goal

- Installing FreeBSD
- Configuring Jail
- Configuring PF Firewall
- Installing Nginx
- Installing LetsEncrypt
- Configuring Nginx
- Configuring SSL
- Configuring Security Headers

10-minute introduction to FreeBSD

- Most FreeBSD-fu works essentially unchanged on other Unix-like operating systems. Often also on Linux.
- FreeBSD is the reference platform for many of the core technologies on the Internet (IPv4, IPv6, TCP, etc...)
- Jails: lightweight virtualisation makes it trivial to isolate processes from each other and from the evils on the Internet
- Cheat-cheat for Linux refugees
  - apt-get / yum / emerge / rpm / [package manager of the week] → pkg
  - init.d / systemd / upstart / [init flavour of the week] → rc.d
    - Starting and stopping services: service [start/stop/restart/reload] service (e.g.: service restart nginx)
  - Third-party software lives in /usr/local, not in /usr
  - Default shell is csh, not bash. Tomayto / tomahto.

pf configuration

- pf syntax is intuitive. Generally:
  - [verb] [direction] [interface] [src/dst] [modifiers]
  - e.g.:
    - pass in on re0 from any to any
    - pass in on re0 proto tcp from any to any port 80
    - block out inet6
    - …
- Hardening webservers
  - Allow http and https inbound
  - Block all outbound (web frameworks tend to break)
  - **MUST** block SMTP out!

Installing Nginx

```
pkg install nginx
```

- Configuration in /usr/local/etc/nginx
- Reload: service nginx reload
- Restart: service nginx restart
- Enabling by default sysrc ...

Encryption

- SSL(Secure Socket Layer) Certificate
- Let’s Encrypt
  - Let’s Encrypt is a free, automated, and open certificate authority brought to you by the non-profit Internet Security Research Group (ISRG)
  - Works like another Certificate Authority
  - Compatible with most of the Browsers
  - Major sponsors include mojilla, Cisco, Akamai, Facebook, DigitalOcean

Let's Encrypt

- Let's Encrypt certificates have a short lifetime
- Automation required to keep certificates “live”
  - Official client from letsencrypt.org [bulky]
  - acme.sh (Bourne sh)
  - Acme-tiny.py (Python)
  - Many others, this workshop uses acme-client
- Often as simple as a cron(8) task, can be made as complex as necessary
- **IMPORTANT:** you must renew your certificates!

Encryption - Demystified

- Private Key and Certificate
- Use 2048-Bit Private Keys
  - Longer keys increase complexity without measurably improving security
  - 2048 provides best security/convenience tradeoff
  - 4096 becoming increasingly common
- Protect Private Keys
- Ensure Sufficient Hostname Coverage

Installing acme-client/LetsEncrypt

```
pkg install acme-client
```

INSERT THIS WITHIN NGINX.CONF SERVER BLOCK LISTENING FOR HTTP (ON PORT 80)

```
# Lets encrypt
location ^~ /.well-known/acme-challenge/ {
alias /usr/local/www/acme/;
}
```

REMAINDER OF NGINX CONFIG FOR WEBSITE HERE

Reload nginx with:

```
service nginx reload
```

Create and Sign certificate

```
acme-client -vNn example.com www.example.com
```

-n

Create a new 4096-bit RSA account key if one does not already exist.

-N

Create a new 4096-bit RSA domain key if one does not already exist.

-v

Verbose operation. Specify twice to also trace communication and data transfers.

example.com and www.example.com are the domains for which the certificates should be valid

Configure Nginx

```
server {
                listen          80;
                server_name     example.com www.example.com;
                # Lets encrypt
                location ^~ /.well-known/acme-challenge/ {
                        alias   /usr/local/www/acme/;
                }
                # Redirect other HTTP connections to HTTPS
                location / {
                        return  301 https://example.com$request_uri;
                }
        }
server {
                listen                  443 ssl;
                server_name             example.com www.example.com;
                ssl_certificate         /usr/local/etc/ssl/acme/fullchain.pem;
                ssl_certificate_key     /usr/local/etc/ssl/acme/private/privkey.pem;
                ... MOVE REMAINDER OF NGINX CONFIG FOR WEBSITE HERE
        }
```

Restart nginx with:

```
# service nginx restart
```

Configure periodic re-validation of certificate

```
weekly_acme_client_enable="YES"
# To specify the domain name(s) to include in the certificate
weekly_acme_client_domains="example.com www.example.com"

#To specify the .well-known/acme-challenge directory (full path)
weekly_acme_client_challengedir="/usr/local/www/acme"

#To set additional acme-client arguments (see acme-client(1))
weekly_acme_client_args="-bv"

#To run a specific script for the renewal (ignore previously set variables)
#allows generating/renewing multiple keys/certificates
#weekly_acme_client_renewscript="/usr/local/etc/acme/acme-client.sh"

#To run a script after the renewal to deploy changed certs
#weekly_acme_client_deployscript="/usr/local/etc/acme/deploy.sh"
```

Test Re-Validation

```
/usr/local/etc/periodic/weekly/000.
acme-client.sh
```

But how secure ?

https://github.com/mozilla/tls-observatory

https://github.com/mozilla/cipherscan

https://www.ssllabs.com/ssltest/

LetsSecure

- Use http/2 instead of http/1.1

```
listen 443 ssl http2;
listen [::]:443 ssl http2;
```

Encryption – HTTP Strict Transport Security

- Sites have always heavily relied on a 301/302 redirect to take users from browsing over HTTP to HTTPS. With browsers defaulting to HTTP when you type in an address, this has previously been the only way. HSTS allows you to tell a browser that you always want a user to connect using HTTPS instead of HTTP. This means any bookmarks, links or addresses the user types will be forced to use HTTPS, even if they specify HTTP.
- When possible, you should enable HTTP Strict Transport Security (HSTS), which instructs browsers to communicate with your site only over HTTPS.

LetsSecure

- Send header to tell the browser to prefer https to http traffic

```
add_header Strict-Transport-
Security max-age=31536000;
```

SSL – The BEAST Attack

- By tampering with an encryption algorithm's CBC - cipher block chaining - mode's, portions of the encrypted traffic can be secretly decrypted.
- Recent browser versions have enabled client side mitigation for the beast attack. The recommendation was to disable all TLS 1.0 ciphers and only offer RC4. However, [RC4 has a growing list of attacks against it],(http://www.isg.rhul.ac.uk/tls/) many of which have crossed the line from theoretical to practical. Moreover, there is reason to believe that the NSA has broken RC4, their so-called "big breakthrough."
- Disabling RC4 has several ramifications. One, users with shitty browsers such as Internet Explorer on Windows XP will use 3DES in lieu. Triple-DES is more secure than RC4, but it is significantly more expensive. Your server will pay the cost for these users. Two, RC4 mitigates BEAST. Thus, disabling RC4 makes TLS 1.0 users susceptible to that attack, by moving them to AES-CBC (the usual server-side BEAST "fix" is to prioritize RC4 above all else). I am confident that the flaws in RC4 significantly outweigh the risks from BEAST. Indeed, with client-side mitigation (which Chrome and Firefox both provide), BEAST is a nonissue. But the risk from RC4 only grows: More cryptanalysis will surface over time.

SSL – Factoring RSA-EXPORT Keys

- FREAK is a man-in-the-middle (MITM) vulnerability discovered by a group of cryptographers at INRIA, Microsoft Research and IMDEA. FREAK stands for "Factoring RSA-EXPORT Keys."
- The vulnerability dates back to the 1990s, when the US government banned selling crypto software overseas, unless it used export cipher suites which involved encryption keys no longer than 512-bits.
- It turns out that some modern TLS clients - including Apple's SecureTransport and OpenSSL - have a bug in them. This bug causes them to accept RSA export-grade keys even when the client didn't ask for export-grade RSA. The impact of this bug can be quite nasty: it admits a 'man in the middle' attack whereby an active attacker can force down the quality of a connection, provided that the client is vulnerable and the server supports export RSA.
- There are two parts of the attack as the server must also accept "export grade RSA."
- The MITM attack works as follows:
  - In the client's Hello message, it asks for a standard 'RSA' ciphersuite.
  - The MITM attacker changes this message to ask for 'export RSA'.
  - The server responds with a 512-bit export RSA key, signed with its long-term key.
  - The client accepts this weak key due to the OpenSSL/SecureTransport bug.
  - The attacker factors the RSA modulus to recover the corresponding RSA decryption key.
  - When the client encrypts the 'pre-master secret' to the server, the attacker can now decrypt it to recover the TLS 'master secret'.
  - From here on out, the attacker sees plaintext and can inject anything it wants.
- The ciphersuite offered here on this page does not enable EXPORT grade ciphers. Make sure your OpenSSL is updated to the latest available version and urge your clients to also use upgraded software.

SSL – Logjam

- Researchers from several universities and institutions conducted a study that found an issue in the TLS protocol. In a report the researchers report two attack methods.
- Diffie-Hellman key exchange allows that depend on TLS to agree on a shared key and negotiate a secure session over a plain text connection.
- With the first attack, a man-in-the-middle can downgrade a vulnerable TLS connection to 512-bit export-grade cryptography which would allow the attacker to read and change the data. The second threat is that many servers and use the same prime numbers for Diffie-Hellman key exchange instead of generating their own unique DH parameters.
- The team estimates that an academic team can break 768-bit primes and that a nation-state could break a 1024-bit prime. By breaking one 1024-bit prime, one could eavesdrop on 18 percent of the top one million HTTPS domains. Breaking a second prime would open up 66 percent of VPNs and 26 percent of SSH servers.
- Later on in this guide we generate our own unique DH parameters and we use a ciphersuite that does not enable EXPORT grade ciphers. Make sure your OpenSSL is updated to the latest available version and urge your clients to also use upgraded software. Updated browsers refuse DH parameters lower than 768/1024 bit as a fix to this.

Encryption – Heartbleed

- Heartbleed is a security bug disclosed in April 2014 in the OpenSSL cryptography library, which is a widely used implementation of the Transport Layer Security (TLS) protocol. Heartbleed may be exploited regardless of whether the party using a vulnerable OpenSSL instance for TLS is a server or a client. It results from improper input validation (due to a missing bounds check) in the implementation of the DTLS heartbeat extension (RFC6520), thus the bug's name derives from "heartbeat". The vulnerability is classified as a buffer over-read, a situation where more data can be read than should be allowed.
- What versions of the OpenSSL are affected by Heartbleed?
- Status of different versions:
  - OpenSSL 1.0.1 through 1.0.1f (inclusive) are vulnerable
  - OpenSSL 1.0.1g is NOT vulnerable
  - OpenSSL 1.0.0 branch is NOT vulnerable
  - OpenSSL 0.9.8 branch is NOT vulnerable
- The bug was introduced to OpenSSL in December 2011 and has been out in the wild since OpenSSL release 1.0.1 on 14th of March 2012. OpenSSL 1.0.1g released on 7th of April 2014 fixes the bug.
- By updating OpenSSL you are not vulnerable to this bug.

Encryption – SSL Compression

- The CRIME attack uses SSL Compression to do its magic. SSL compression is turned off by default in nginx 1.1.6+/1.0.9+ (if OpenSSL 1.0.0+ used) and nginx 1.3.2+/1.2.2+ (if older versions of OpenSSL are used).
- If you are using al earlier version of nginx or OpenSSL and your distro has not backported this option then you need to recompile OpenSSL without ZLIB support. This will disable the use of OpenSSL using the DEFLATE compression method. If you do this then you can still use regular HTML DEFLATE compression.

SSL – SSLv2 and SSLv3

- SSL v2 is insecure, so we need to disable it. We also disable SSLv3, as TLS 1.0 suffers a downgrade attack, allowing an attacker to force a connection to use SSLv3 and therefore disable forward secrecy.

SSL – Poodle Attack

- SSLv3 allows exploiting of the POODLE bug. This is one more major reason to disable this.
- Google have proposed an extension to SSL/TLS named TLSFALLBACKSCSV that seeks to prevent forced SSL downgrades. This is automatically enabled if you upgrade OpenSSL to the following versions:
  - OpenSSL 1.0.1 has TLSFALLBACKSCSV in 1.0.1j and higher.
  - OpenSSL 1.0.0 has TLSFALLBACKSCSV in 1.0.0o and higher.
  - OpenSSL 0.9.8 has TLSFALLBACKSCSV in 0.9.8zc and higher.

LetsSecure

- Use TLS instead of SSL - Compatibility issues with some Java clients and older versions of of IE, however, more secure.

```
ssl_protocols TLSv1.1 TLSv1.2;
```

Encryption – Cipher Suite

- Forward Secrecy ensures the integrity of a session key in the event that a long-term key is compromised. PFS accomplishes this by enforcing the derivation of a new key for each and every session.
- This means that when the private key gets compromised it cannot be used to decrypt recorded SSL traffic.
- The cipher suites that provide Perfect Forward Secrecy are those that use an ephemeral form of the Diffie-Hellman key exchange. Their disadvantage is their overhead, which can be improved by using the elliptic curve variants.
- The following two ciphersuites are recommended, and the latter by the Mozilla Foundation.
- The recommended cipher suite:
  - ssl_ciphers 'EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH';
- The recommended cipher suite for backwards compatibility (IE6/WinXP):
  - ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:ECDHE-RSA-AES128-GCM-SHA256:AES256+EECDH:DHE-RSA-AES128-GCM-SHA256:AES256+EDH:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA256:ECDHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:DHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA:ECDHE-RSA-DES-CBC3-SHA:EDH-RSA-DES-CBC3-SHA:AES256-GCM-SHA384:AES128-GCM-SHA256:AES256-SHA256:AES128-SHA256:AES256-SHA:AES128-SHA:DES-CBC3-SHA:HIGH:!aNULL:!eNULL:!EXPORT:!DES:!MD5:!PSK:!RC4";
- If your version of OpenSSL is old, unavailable ciphers will be discarded automatically. Always use the full ciphersuite above and let OpenSSL pick the ones it supports.
- The ordering of a ciphersuite is very important because it decides which algorithms are going to be selected in priority. The recommendation above prioritizes algorithms that provide perfect forward secrecy.
- Older versions of OpenSSL may not return the full list of algorithms. AES-GCM and some ECDHE are fairly recent, and not present on most versions of OpenSSL shipped with Ubuntu or RHEL.

Encryption – Prioritization Logic

- ECDHE+AESGCM ciphers are selected first. These are TLS 1.2 ciphers. No known attack currently target these ciphers.
- PFS ciphersuites are preferred, with ECDHE first, then DHE.
- AES 128 is preferred to AES 256. There has been discussions on whether AES256 extra security was worth the cost, and the result is far from obvious. At the moment, AES128 is preferred, because it provides good security, is really fast, and seems to be more resistant to timing attacks.
- In the backward compatible ciphersuite, AES is preferred to 3DES. BEAST attacks on AES are mitigated in TLS 1.1 and above, and difficult to achieve in TLS 1.0. In the non-backward compatible ciphersuite, 3DES is not present.
- RC4 is removed entirely. 3DES is used for backward compatibility.

Encryption – Mandatory Discards

- aNULL contains non-authenticated Diffie-Hellman key exchanges, that are subject to Man-In-The-Middle (MITM) attacks
- eNULL contains null-encryption ciphers (cleartext)
- EXPORT are legacy weak ciphers that were marked as exportable by US law
- RC4 contains ciphers that use the deprecated ARCFOUR algorithm
- DES contains ciphers that use the deprecated Data Encryption Standard
- SSLv2 contains all ciphers that were defined in the old version of the SSL standard, now deprecated
- MD5 contains all the ciphers that use the deprecated message digest 5 as the hashing algorithm

LetsSecure

- Use more secure and less CPU tasking ciphers compared to nginx defaults

```
ssl_ciphers
ECDH+AESGCM:DH+AESGCM:ECDH+AES256:D
H+AES256:ECDH+AES128:DH+AES:ECDH+3D
ES:DH+3DES:RSA+AESGCM:RSA+AES:RSA+3
DES:!aNULL:!MD5:!DSS;
```

- Specifies that server ciphers should be preferred over client ciphers

```
ssl_prefer_server_ciphers on;
```

Encryption – Forward Secrecy

- The concept of forward secrecy is simple: client and server negotiate a key that never hits the wire, and is destroyed at the end of the session. The RSA private from the server is used to sign a Diffie-Hellman key exchange between the client and the server. The pre-master key obtained from the Diffie-Hellman handshake is then used for encryption. Since the pre-master key is specific to a connection between a client and a server, and used only for a limited amount of time, it is called Ephemeral.
- With Forward Secrecy, if an attacker gets a hold of the server's private key, it will not be able to decrypt past communications. The private key is only used to sign the DH handshake, which does not reveal the pre-master key. Diffie-Hellman ensures that the pre-master keys never leave the client and the server, and cannot be intercepted by a MITM.
- All versions of nginx as of 1.4.4 rely on OpenSSL for input parameters to Diffie-Hellman (DH). Unfortunately, this means that Ephemeral Diffie-Hellman (DHE) will use OpenSSL's defaults, which include a 1024-bit key for the key-exchange. Since we're using a 2048-bit certificate, DHE clients will use a weaker key-exchange than non-ephemeral DH clients.

LetsSecure

- Specifies a file with DH parameters for EDH ciphers
- Run "`openssl dhparam -out /path/to/ssl/dhparam.pem 4096`" in terminal to generate it

```
ssl_dhparam
/path/to/ssl/dhparam.pem;
```

LetsSecure

- Improves TTFB by using a smaller SSL buffer than the nginx default

```
ssl_buffer_size 8k;
```

- Enables all nginx worker processes share SSL session information

```
ssl_session_cache shared:SSL:30m;
```

- Increases the amount of time SSL session information in the cache is valid

```
ssl_session_timeout 30m;
```

- File containing chain of domain and intermediate certificates

```
ssl_certificate /path/to/ssl/domain-intermediate-cert.crt;
```

- File containing private key

```
ssl_certificate_key /path/to/ssl/private.key;
```

Encryption – OCSP Stapling

- When connecting to a server, clients should verify the validity of the server certificate using either a Certificate Revocation List (CRL), or an Online Certificate Status Protocol (OCSP) record. The problem with CRL is that the lists have grown huge and takes forever to download.
- OCSP is much more lightweight, as only one record is retrieved at a time. But the side effect is that OCSP requests must be made to a 3rd party OCSP responder when connecting to a server, which adds latency and potential failures. In fact, the OCSP responders operated by CAs are often so unreliable that browser will fail silently if no response is received in a timely manner. This reduces security, by allowing an attacker to DoS an OCSP responder to disable the validation.
- The solution is to allow the server to send its cached OCSP record during the TLS handshake, therefore bypassing the OCSP responder. This mechanism saves a roundtrip between the client and the OCSP responder, and is called OCSP Stapling.
- The server will send a cached OCSP response only if the client requests it, by announcing support for the status_request TLS extension in its CLIENT HELLO.
- Most servers will cache OCSP response for up to 48 hours. At regular intervals, the server will connect to the OCSP responder of the CA to retrieve a fresh OCSP record. The location of the OCSP responder is taken from the Authority Information Access field of the signed certificate.

LetsSecure

- Enables OCSP(Online Certificate Status Protocol) stapling

```
ssl_stapling on;

ssl_stapling_verify on;

resolver 8.8.8.8 8.8.4.4
valid=300s;

ssl_trusted_certificate
/usr/local/etc/ssl/example.com.trus
ted.pem;
```

Encryption – HTTP Public Key Pinning

- We should also enable the HTTP Public Key Pinning Extension.
- Public Key Pinning means that a certificate chain must include a whitelisted public key. It ensures only whitelisted Certificate Authorities (CA) can sign certificates for *.example.com, and not any CA in your browser store.

HTTP Public Key Pinning

- The great thing about SSL/TLS certificates is that you can buy a certificate from any trusted Certificate Authority and the browser will happily accept it. The problem with this is that when a Certificate Authority is compromised, an attacker can issue themselves an SSL/TLS certificate for your site and the browser will accept this rogue certificate as it came from a trusted Certificate Authority. HPKP allows you to protect yourself by providing a whitelist of cryptographic identities that the browser should trust. Whilst HSTS says the browser must always use HTTPS, HPKP says the browser should only ever accept a specific set of certificates.

HTTP Public Key Pinning - Cont

- The first step to creating a HPKP policy is to get the fingerprint of your current certificate.

```
openssl x509 -pubkey < tls.crt | openssl pkey -pubin
-outform der | openssl dgst -sha256 -binary | base64
```

- Creating A Backup CSR

```
openssl genrsa -out bofh1.key 4096
```

```
openssl req -new -key bofh1.key -sha256 -out bofh1.csr
```

- Fingerprint for Backup CSR

```
openssl req -pubkey < bofh1.csr | openssl pkey -pubin
-outform der | openssl dgst -sha256 -binary | base64
```

HTTP Public Key Pinning - Cont

- Creating A Backup of Backup CSR

```
openssl genrsa -out bofh2.key 4096

openssl req -new -key bofh2.key -sha256 -out bofh2.csr
```

- Fingerprint for Backup Backup CSR

```
openssl req -pubkey < bofh2.csr | openssl pkey -pubin
-outform der | openssl dgst -sha256 -binary | base64
```

- Configure Nginx

```
add_header Public-Key-Pins 'pin-
sha256="X3pGTSoUGeEGw583IJ/eEtXUEmy52zs1TZQrU06KUKg="; \

pin-sha256="MHJYVThihUrJcxW7wcqyOISTXIsInsdj3xK8QrZbHec="; \

pin-sha256="isi41AizREkLvvfr0IRW4u3XMFR2Yg7bvrF7padyCJg="; \

max-age=10';
```

Encryption – 2048-bit Private Keys

- For most web sites, security provided by 2,048-bit RSA keys is sufficient.
- The RSA public key algorithm is widely supported, which makes keys of this type a safe default choice. At 2,048 bits, such keys provide about 112 bits of security. If you want more security than this, note that RSA keys don't scale very well. To get 128 bits of security, you need 3,072-bit RSA keys, which are noticeably slower.
- ECDSA keys provide an alternative that offers better security and better performance. At 256 bits, ECDSA keys provide 128 bits of security. A small number of older clients don't support ECDSA, but modern clients do. It's possible to get the best of both worlds and deploy with RSA and ECDSA keys simultaneously if you don't mind the overhead of managing such a setup.

Encryption – Protect Private Keys

- Treat your private keys as an important asset, restricting access to the smallest possible group of employees while still keeping your arrangements practical. Recommended policies include the following:
- Generate private keys on a trusted computer with sufficient entropy. Some CAs offer to generate private keys for you; run away from them.
- Password-protect keys from the start to prevent compromise when they are stored in backup systems. Private key passwords don’t help much in production because a knowledgeable attacker can always retrieve the keys from process memory. There are hardware devices (called Hardware Security Modules, or HSMs) that can protect private keys even in the case of server compromise, but they are expensive and thus justifiable only for organizations with strict security requirements.
- After compromise, revoke old certificates and generate new keys.
- Renew certificates yearly, and more often if you can automate the process. Most sites should assume that a compromised certificate will be impossible to revoke reliably; certificates with shorter lifespans are therefore more secure in practice.
- Unless keeping the same keys is important for public key pinning, you should also generate new private keys whenever you're getting a new certificate.

Encryption – Sufficient Hostname Coverage

- Ensure that your certificates cover all the names you wish to use with a site. Your goal is to avoid invalid certificate warnings, which confuse users and weaken their confidence.
- Even when you expect to use only one domain name, remember that you cannot control how your users arrive at the site or how others link to it. In most cases, you should ensure that the certificate works with and without the www prefix (e.g., that it works for both example.com and www.example.com). The rule of thumb is that a secure web server should have a certificate that is valid for every DNS name configured to point to it.
- Wildcard certificates have their uses, but avoid using them if it means exposing the underlying keys to a much larger group of people, and especially if doing so crosses team or department boundaries. In other words, the fewer people there are with access to the private keys, the better. Also be aware that certificate sharing creates a bond that can be abused to transfer vulnerabilities from one web site or server to all other sites and servers that use the same certificate (even when the underlying private keys are different).

Encryption – Certificate Signature Algorithm

- Certificate security depends
  - On the strength of the private key that was used to sign the certificate
  - The strength of the hashing function used in the signature.
- Until recently, most certificates relied on the SHA1 hashing function, which is now considered insecure. As a result, we're currently in transition to SHA256. As of January 2016, you shouldn't be able to get a SHA1 certificate from a public CA. The existing SHA1 certificates will continue to work (with warnings in some browsers), but only until the end of 2016.

Encryption - Demystified

- Configuration
  - Complete Certificate Chains
  - Secure Protocols
  - Secure Cipher Suites
  - Forward Secrecy
  - Strong Key Exchange
  - Mitigate Known Problems

Configuration – Complete Certificate Chains

- In most deployments, the server certificate alone is insufficient; two or more certificates are needed to build a complete chain of trust. A common configuration problem occurs when deploying a server with a valid certificate, but without all the necessary intermediate certificates. To avoid this situation, simply use all the certificates provided to you by your CA.
- An invalid certificate chain effectively renders the server certificate invalid and results in browser warnings. In practice, this problem is sometimes difficult to diagnose because some browsers can reconstruct incomplete chains and some can’t. All browsers tend to cache and reuse intermediate certificates.

Configuration – Secure Protocol

- There are five protocols in the SSL/TLS family: SSL v2, SSL v3, TLS v1.0, TLS v1.1, and TLS v1.2:
- SSL v2 is insecure and must not be used. This protocol version is so bad that it can be used to attack RSA keys and sites with the same name even if they are on an entirely different servers (the DROWN attack).
- SSL v3 is insecure when used with HTTP (the POODLE attack) and weak when used with other protocols. It’s also obsolete and shouldn’t be used.
- TLS v1.0 is also a legacy protocol that shouldn't be used, but it's typically still necessary in practice. Its major weakness (BEAST) has been mitigated in modern browsers, but other problems remain.
- TLS v1.1 and v1.2 are both without known security issues, but only v1.2 provides modern cryptographic algorithms.

Configuration – Secure Cipher Suites

- To communicate securely, you must first ascertain that you are communicating directly with the desired party (and not through someone else who will eavesdrop) and exchanging data securely. In SSL and TLS, cipher suites define how secure communication takes place. They are composed from varying building blocks with the idea of achieving security through diversity. If one of the building blocks is found to be weak or insecure, you should be able to switch to another.
- You should rely chiefly on the AEAD suites that provide strong authentication and key exchange, forward secrecy, and encryption of at least 128 bits. Some other, weaker suites may still be supported, provided they are negotiated only with older clients that don't support anything better.

Configuration – Forward Secrecy

- Forward secrecy (sometimes also called perfect forward secrecy) is a protocol feature that enables secure conversations that are not dependent on the server's private key. With cipher suites that do not provide forward secrecy, someone who can recover a server's private key can decrypt all earlier recorded encrypted conversations. You need to support and prefer ECDHE suites in order to enable forward secrecy with modern web browsers. To support a wider range of clients, you should also use DHE suites as fallback after ECDHE. Avoid the RSA key exchange unless absolutely necessary.

Configuration – Strong Key Exchange

- For the key exchange, public sites can typically choose between the classic ephemeral Diffie-Hellman key exchange (DHE) and its elliptic curve variant, ECDHE. There are other key exchange algorithms, but they're generally insecure in one way or another. The RSA key exchange is still very popular, but it doesn't provide forward secrecy.
- In 2015, a group of researchers published new attacks against DHE; their work is known as the Logjam attack. The researchers discovered that lower-strength DH key exchanges (e.g., 768 bits) can easily be broken and that some well-known 1,024-bit DH groups can be broken by state agencies. To be on the safe side, if deploying DHE, configure it with at least 2,048 bits of security. Some older clients (e.g., Java 6) might not support this level of strength. For performance reasons, most servers should prefer ECDHE, which is both stronger and faster. The secp256r1 named curve (also known as P-256) is a good choice in this case.

Content-Security-Policy

- The CSP header allows you to define a whitelist of approved sources of content for your site. By restricting the assets that a browser can load for your site, like js and css, CSP can act as an effective countermeasure to XSS attacks. Here is a basic policy to enforce TLS on all assets and prevent mixed content warnings.

Configure - Content-Security-Policy

```
add_header Content-Security-Policy
"default-src https: data: 'unsafe-
inline' 'unsafe-eval'" always;
```

X-Frame-Options

- The X-Frame-Options header (RFC), or XFO header, protects your visitors against clickjacking attacks. An attacker can load up an iframe on their site and set your site as the source, it's quite easy: <iframe src="https://example.com"></iframe>. Using some crafty CSS they can hide your site in the background and create some genuine looking overlays. When your visitors click on what they think is a harmless link, they're actually clicking on links on your website in the background. That might not seem so bad until we realize that the browser will execute those requests in the context of the user, which could include them being logged in and authenticated to your site! Troy Hunt has a great blog on Clickjack attack – the hidden threat right in front of you. Valid values include DENY meaning your site can't be framed, SAMEORIGIN which allows you to frame your own site or ALLOW-FROM https://example.com/ which lets you specify sites that are permitted to frame your own site.

```
add_header X-Frame-Options "SAMEORIGIN" always;
```

X-Xss-Protection

- This header is used to configure the built in reflective XSS protection found in Internet Explorer, Chrome and Safari (Webkit). Valid settings for the header are 0, which disables the protection, 1 which enables the protection and 1; mode=block which tells the browser to block the response if it detects an attack rather than sanitising the script.

```
add_header X-Xss-Protection "1; mode=block" always;
```

X-Content-Type-Options

- This header only has one valid value, nosniff. It prevents Google Chrome and Internet Explorer from trying to mime-sniff the content-type of a response away from the one being declared by the server. It reduces exposure to drive-by downloads and the risks of user uploaded content that, with clever naming, could be treated as a different content-type, like an executable.

```
add_header X-Content-Type-Options
"nosniff" always;
```

Hands on Lab

- Hands on Lab preferably on FreeBSD/Nginx