Capítulo 5

Introdução à criptografia para administradores de sistemas com TLS, `OpenSSL` e Apache `mod_ssl`

Alexandre Braga (UNICAMP) e Ricardo Dahab (UNICAMP)

Abstract

The incorrect use of cryptographic infrastructures is one of the most common sources of cryptography-related security issues: avoidable errors related to configuration of algorithms and protocols become recurrent, often resulting in exploitable vulnerabilities and actual attacks on computing systems. Hence, there is a growing demand for practical, real-world guidance on how to avoid incorrect cryptographic configurations that lead to exploitable vulnerabilities in computer networks and distributed systems. This chapter addresses the use of cryptography by students and IT professionals with little experience in information security and cryptography. The text covers `OpenSSL`, the TLS protocol (including the new version 1.3 launched in August 2018), and security settings for the `mod_ssl` module integrated to the Apache web server.

Resumo

O uso incorreto de infraestruturas criptográficas é uma das fontes mais comuns de problemas de segurança relacionados à criptografia, quando diversos erros evitáveis na configuração de algoritmos e de protocolos se tornam recorrentes, resultando frequentemente em vulnerabilidades exploráveis em ataques reais aos sistemas de computação. Por isso, há uma demanda crescente por orientações práticas, do mundo real, sobre como evitar configurações incorretas da criptografia que levam a vulnerabilidades exploráveis em redes de computadores e sistemas distribuídos. Este capítulo aborda a utilização de criptografia por estudantes e profissionais de TI com pouca experiência em segurança da informação e criptografia. O minicurso cobre o `OpenSSL`, o protocolo TLS (incluindo a nova versão 1.3 de agosto de 2018) e as configurações seguras do módulo `mod_ssl` do servidor web Apache.

5.1. Introdução

O uso incorreto de infraestruturas criptográficas é uma das fontes mais comuns de problemas de segurança relacionados à criptografia, em que diversos erros evitáveis na configuração de algoritmos e de protocolos se tornaram recorrentes, resultando frequentemente em vulnerabilidades exploráveis em ataques reais aos sistemas de computação. No contexto da indústria de Tecnologia da Informação (TI), há uma demanda crescente pelos aspectos práticos, do mundo real, relacionados às configurações corretas da criptografia que evitem vulnerabilidades exploráveis em redes de computadores e sistemas distribuídos.

Este capítulo cobre, em nível introdutório, o `OpenSSL`, o protocolo TLS, incluindo a nova versão (v1.3) de agosto de 2018, e as configurações do módulo `mod_ssl` do servidor web Apache. O software `OpenSSL` é a principal ferramenta de criptografia escolhida para a realização de exercícios e demonstrações por que possui uma *Command Line Interface* (CLI) estável e que tem sido usada por profissionais de TI há muito tempo. Além disso, o protocolo TLS e o `OpenSSL` compõem a infraestrutura criptográfica utilizada por diversos softwares na base dos sistemas da Internet, tais como servidores web (incluindo o `mod_ssl` do Apache), servidores de aplicação, contêineres de software, plataformas móveis e até dispositivos da Internet das Coisas, entre outros.

O restante do texto está organizado da seguinte forma. A Seção 5.2 conta uma breve história da evolução do SSL, oferecendo uma visão geral do funcionamento deste protocolo. A Seção 5.3 recapitula os conceitos fundamentais da criptografia. A Seção 5.4 está organizada como um tutorial de comandos do `OpenSSL`. Já a Seção 5.5 mostra como testar a segurança criptográfica do servidor Apache, enquanto a Seção 5.6 descreve as configurações seguras e inseguras de SSL. Finalmente, a Seção 5.7 faz as considerações finais.

5.2. Breve história do surgimento e evolução do SSL

Security Sockets Layer, ou simplesmente SSL (camada de segurança de *sockets*, em tradução livre) é o protocolo para transporte de dados protegidos com criptografia sobre os *sockets* do protocolo TCP. Em relação a pilha de protocolos TCP/IP, a camada de *socket* seguro fica logo acima da camada de transporte. De fato, as aplicações (na camada de aplicação) podem usar os *sockets* seguros como se fossem um serviço oferecido pela camada de transporte.

O SSL permite que a comunicação pela Internet entre aplicações cliente/servidor possa ser protegida contra monitoramento, adulteração de conteúdo e falsificação de mensagens. O objetivo principal do SSL sempre foi proporcionar um canal de comunicação seguro entre partes comunicantes. Tradicionalmente, livros de segurança em redes de computadores [Stallings 2003] abordam o SSL, havendo também livros específicos [Chandra et al. 2002, Ristic 2015] sobre a implementação de código aberto mais conhecida deste protocolo, o `OpenSSL` [OpenSSL.org].

A história do protocolo SSL se confunde com a história do comércio eletrônico na Internet. Na última década do século passado, a empresa Netscape [Wikipedia 2018] dominava tanto o mercado de software de navegadores web (*browsers*) quanto o de servidores Web. Em um modelo de negócios inovador para a época, a Netscape distribuía gratuitamente o *browser*, mas vendia o seu servidor web, o único na época que possuia proteções de sigilo contra monitoramento e interceptação da comunicação. A segurança da comunicação era garantida por

um protocolo criptográfico na camada de transporte do TCP/IP que ficou conhecido como SSL.

O SSL tornou possíveis as transações sigilosas entre o*browser* web na máquina do usuário e o servidor web. O uso do SSL sempre foi praticamente transparente para o usuário final, o que facilitou muito a ampla adoção deste protocolo como padrão de fato para segurança na Internet. Por exemplo, com o SSL, os números de cartões de crédito puderam transitar pelas redes abertas com sigilo, viabilizando o comércio eletrônico como acontece hoje em dia. A última versão do SSL original foi a terceira (SSLv3), que trouxe as seguintes características de segurança disponíveis até hoje:

- Autenticação do servidor com assinaturas digitais e certificados X.509, com os algoritmos RSA, DSA e ECDSA, ou uma chave simétrica pré-configurada;
- Autenticação (opcional) do cliente com assinaturas digitais e certificados X.509;
- Sigilo com encriptação da informação trocada entre cliente e servidor;
- Integridade (com uso de MACs) da informação trocada entre cliente e servidor.

Uma vez que o SSL se tornou um padrão de fato, com o seu uso bastante difundido, surgiu a necessidade de padronização da tecnologia. Foi a terceira versão do SSL (SSLv3) que serviu de base para o *Transport Layer Security* (TLS), um protocolo padronizado pelo *Internet Engineering Task Force* (IETF) com o objetivo de substituir o formato proprietário do SSL original. Por razões históricas, o protocolo TLS também é conhecido como SSL/TLS.

O SSL/TLS possui dois sub-protocolos: a saudação (*handshake*), que autentica as partes e negocia os parâmetros de segurança criptográfica da comunicação, e o protocolo de registro, que usa os parâmetros escolhidos na saudação para proteger o tráfego entre as partes comunicantes. A Figura 5.1 ilustra o funcionamento do *handshake* do SSL/TLS v1.2. As etapas do diálogo de saudação entre cliente e servidor são detalhadas e enumeradas na Figura 5.1. O objetivo principal deste diálogo é o estabelecimento de uma chave de sessão (uma chave criptográfica secreta, simétrica e temporária). O diálogo de saudação pode ser dividido em quatro partes:

1. Negociação de parâmetros (versão, ID de sessão, algoritmos criptográficos e compressão);
2. Envio do certificado do servidor e solicitação opcional do certificado do cliente;
3. Envio do certificado do cliente, se solicitado;
4. Escolha de algoritmos criptográficos e finalização;

A implementação do *handshake* em duas rodadas é considerada [Sullivan 2018] uma causa de perda de desempenho na execução do protocolo TLSv1.2 e foi otimizada na versão TLSv1.3. Após a realização do protocolo de *handshake*, entra em ação o protocolo de registro que realiza a troca, entre cliente e servidor, de informação encriptada e segmentada em blocos chamados de registros.

5.2.1. Versões do SSL/TLS e suas vulnerabilidades

Além do SSLv3 que deu origem ao TLSv1, vale ainda mencionar outras versões do SSL/TLS que se destacam por possuírem características específicas e vulnerabilidades conhecidas. Em linhas gerais, existem seis versões em uso do SSL/TLS que ainda podem ser encontradas em

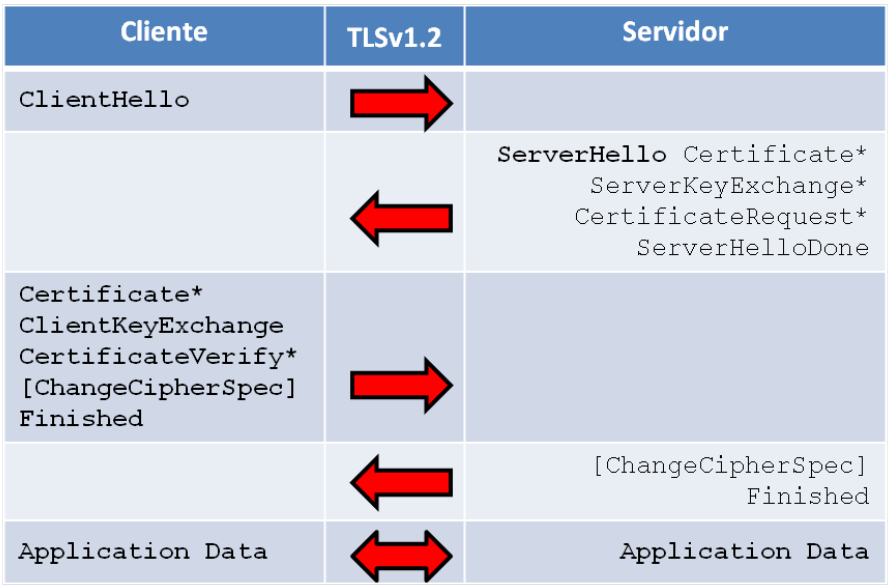


Figura 5.1. Estabelecimento de sessão e envio de mensagens SSL/TLS v1.2.

implantações reais: SSLv2, SSLv3, TLSv1.0, TLSv1.1, TLSv1.2 e TLSv1.3. As características gerais de cada uma delas são descritas a seguir:

- SSLv2 é considerado inseguro e não deve mais ser usado. Esta versão é vulnerável a ataques conhecidos, tais como o BEAST (Browser Exploit against SSL/TLS) e o DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) [Aviram et al. 2016], e outras vulnerabilidades como *Cipher Suite Rollback* e *ChangeCipherSpec Message Drop*.
- SSLv3 é obsoleto e não deve mais ser utilizado. O HTTPS sobre SSLv3 é vulnerável aos ataques POODLE (Padding Oracle on Downgraded Legacy Encryption) [Möller et al. 2014] e *Lucky 13* [Fardan and Paterson 2013] e sofre de vulnerabilidades como o *Version Rollback* e o *Key Exchange Algorithm Confusion*.
- TLSv1.0 (RFC 2246) ainda é usada em sistemas legados. TLSv1.0 é vulnerável aos ataques BEAST e *Lucky 13* [Fardan and Paterson 2013]. Esta versão inicial do TLS também é vulnerável aos ataques de negação de serviço e que exploram falhas na renegociação. Padrões de segurança como o PCI-DSS recomendam que esta versão seja abandonada.
- TLSv1.1 (RFC 4346) é uma versão relativamente recente e que não apresenta vulnerabilidades conhecidas sem mitigação, mas ainda oferece algoritmos criptográficos antigos.
- TLSv1.2 (RFC 5246) era a versão atual até Agosto de 2018, quando foi lançada a versão nova, e já oferece esquemas criptográficos novos com encriptação autenticada.
- TLSv1.3 (RFC 8446) foi lançado na forma final em Agosto de 2018 [Rescorla 2018] e representa o rompimento definitivo com o passado pela eliminação de diversas características inseguras ou obsoletas mantidas até a versão anterior por compatibilidade com legados.

Há testes específicos para implantações do SSL/TLS [Ristic 2015, Eldewahi et al. 2015, OWASP 2015]. Outros ataques contra SSL/TLS bastante conhecidos são o CRIME (*Compression Ratio info-leak Made Easy*)[CRI 2012], o BREACH (*Browser Reconnaissance and*

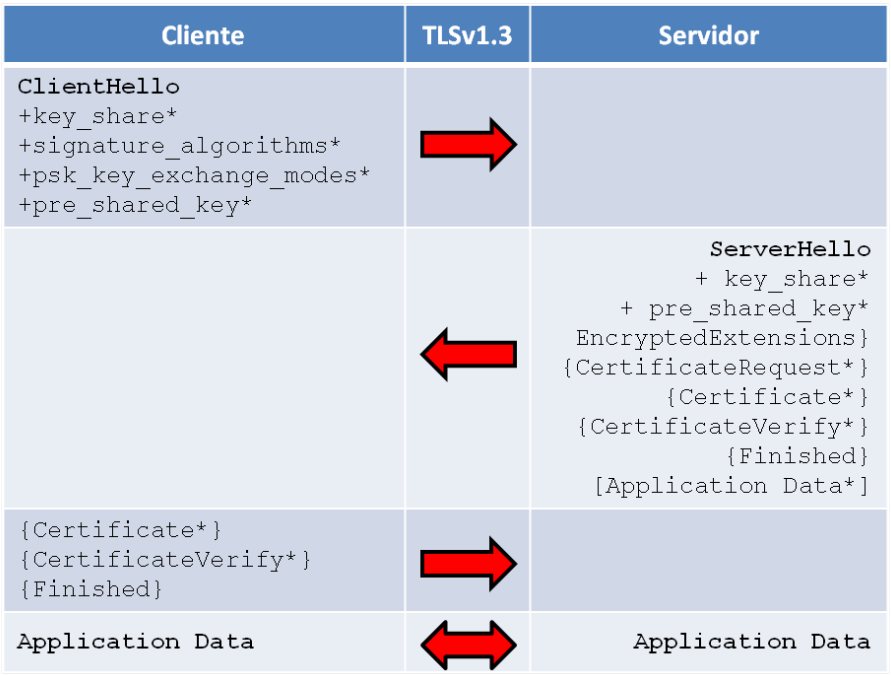


Figura 5.2. Estabelecimento de sessão e envio de mensagens SSL/TLS v1.3.

Exfiltration via Adaptive Compression of Hypertext) [BRE 2013, Gluck et al. 2013], o *Bleichenbacher Attack on PKCS#l* [Bleichenbacher 1998], os ataques ao RC4 [AlFardan et al. 2013], o Heartbleed [Synopsys 2014], e o LogJam [Adrian et al. 2015, Log 2016], que afeta o acordo de chaves.

5.2.2. A versão 1.3 do SSL/TLS

A versão 1.3 do TLS [Rescorla 2018] foi lançada na forma final em Agosto de 2018. Este texto inclui um breve relato dos avanços apresentados por esta nova versão em relação a sua antecessora. A nova versão 1.3 atende a quatro objetivos de projeto [Sullivan 2018]: redução da latência do *handshake*, encriptação de partes do *handshake*, aumento da robustez contra ataques e remoção de funções legadas. A Figura 5.2 ilustra o *handshake* do SSL/TLS na versão 1.3, onde se observa que o protocolo é mais curto que o da versão anterior. As principais diferenças entre o TLSv1.2 e o TLSv1.3 são as seguintes [Rescorla 2018]:

- A lista de opções de algoritmos simétricos foi bastante reduzida e só contém encriptação autenticada (*Authenticated Encryption with Additional Data* - AEAD), já a encriptação de blocos em modo CBC e o RC4 foram removidos da versão;
- O conceito de *suite* criptográfica foi simplificado e agora separa os mecanismos de autenticação e troca de chaves dos mecanismos de encriptação, *hash* usado na derivação de chaves e MACs;
- O *handshake* simplificado, em uma única rodada, acelera a saudação; além disso, o *handshake* sem rodadas (sem negociações, só comunicação de escolhas pré-definidas) foi incluído para atender às aplicações com restrições de tempo ainda maiores;

- RSA e DH com parâmetros estáticos foram removidos, de modo que todas as *suites* criptográficas desta versão possuem *forward secrecy*; porém, só o DHE/ECDHE é usado para troca de chaves;
- Todas as mensagens do *handshake* depois do "ServerHello"agora são encriptadas e assinadas pelo servidor, para proteção da comunicação contra ataques práticos, como o FREAK e o LogJam, e outros considerados teóricos, como a troca maliciosa de curvas elípticas (*curveswap*);
- A criptografia de curvas elípticas passou a fazer parte da especificação principal protocolo, mas a negociação de formatos de representação de pontos da curva foi removida, para incluir apenas um formato não negociável;
- Em termos de criptografia de chaves públicas, o RSA-PSS é o único *padding* do RSA disponível, já o *padding* inseguro do padrão PKCS#1 v1.5 foi removido, assim como também foram removidos o DSA e a compressão;
- DH com parâmetros efêmeros customizados não está mais disponível nesta versão, quer possui apenas um número fixo de opções seguras, evitando a parametrização fraca do DH que leva, por exemplo, ao ataque LogJam [Log 2016];
- A Mensagem "ChangeCipherSpec"foi removida do *handshake*, simplificando o protocolo e aumentando a robustez contra ataques, enquanto o *handshake* com chave pré-compartilhada (*pre-shared key* - PSK) foi simplificado e aumenta a robustez contra ataques.

5.2.3. Limitações do SSL/TLS

O SSL/TLS, como todo protocolo de segurança, tem limitações e não resolve todos os problemas de segurança de transações eletrônicas na Internet. Em particular, o SSL/TLS só protege a comunicação entre o software navegador web e o servidor HTTPS. Por exemplo, uma vez que a informação encriptada de um cartão de crédito chega ao servidor HTTPS do lojista, ela precisa ser decriptada para que o pagamento seja efetivado junto à instituição financeira (banco ou operadora de cartão de crédito). Neste momento, as informações do cartão ficam expostas ao lojista e aos seus funcionários. O SSL/TLS não resolve este problema, pois ele trata apenas da comunicação entre *browser* e servidor web. Outros protocolos de segurança devem ser usados para proteger as informações de pagamento.

Além disso, o SSL/TLS é independente de aplicação e, por isto, sua documentação não especifica como ele deve ser adicionado a um protocolo de aplicação. Assim sendo, decisões de projeto importantes são deixadas a cargo dos implementadores de protocolos de aplicação, tais como em que momento o *handshake* deve ser iniciado ou como interpretar a autenticação dos certificados digitais trocados entre as partes. Esta situação abre espaço para a inclusão de defeitos de implementação que levam a vulnerabilidades exploráveis.

Finalmente, surgiram recentemente ataques que exploram a confiança já estabelecida em *sites* web que possuem conexões SSL/TLS legítimas. O abuso da confiança depositada por usuários na conexão SSL/TLS entre *browser* e servidor web ocorre quando uma conexão SSL/TLS legítima é usada em ataques. Atualmente, mesmo sites web maliciosos, utilizados, por exemplo, em ataques de *phishing*, podem ter certificados SSL legítimos em seus servidores. Isto se deve ao fato de autoridades certificadores emitirem, de forma simplificada e na Internet, certificados

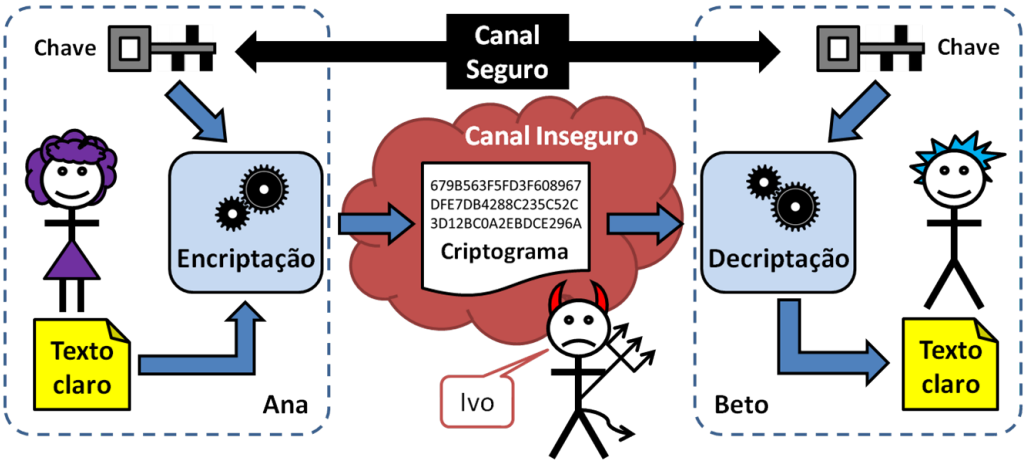


Figura 5.3. Sistema criptográfico (de [Braga and Dahab 2015]).

gratuitos e de validade curta. Estes certificados, mesmo com validade reduzida, ainda podem ser usados por tempo suficiente para viabilizar ataques de engenharia social [Calvo 2018].

5.3. Conceitos fundamentais de criptografia

Historicamente associada ao sigilo, a criptografia moderna também oferece serviços para autenticidade, integridade e irrefutabilidade. Esta seção recapitula os conceitos de serviços criptográficos comumente encontrados em redes de computadores e sistemas de TI. Em linhas gerais, os objetivos e funções da criptografia são os seguintes:

1. Confidencialidade (ou sigilo) é obtida com o uso da criptografia para manter a informação secreta, confidencial. Protocolos de comunicação segura com encriptação de dados em trânsito são exemplos de confidencialidade.
2. Autenticidade é obtida com o uso da criptografia para validar a autoria de uma informação. Um exemplo de autenticação é o uso de assinaturas digitais para verificar a autoria de uma mensagem de texto ou de um documento eletrônico.
3. Integridade é obtida com o uso da criptografia para garantir que o dado não foi modificado desde a sua criação. Valores *hash* usados na validação de arquivos binários são exemplos de mecanismos para verificação de integridade de dados.
4. Irrefutabilidade ou não-repudiação é obtida pelo uso da criptografia para garantir que o autor da informação autêntica não possa negar para um terceiro a autoria desta informação. Assinaturas digitais podem ser usadas na garantia de irrefutabilidade.

Na prática, os objetivos são combinados e as funções são usadas juntas. Por exemplo, em um aplicativo de mensageiria instantânea, a troca de mensagens entre as partes comunicantes pode ser encriptada e assinada digitalmente, garantindo tanto a confidencialidade quanto a autenticidade e a integridade do diálogo.

A Figura 5.3 (adaptada de [Braga and Dahab 2015]) mostra um sistema criptográfico e seus elementos estruturais com três personagens: Ana, a remetente das mensagens; Beto, o destinatário das mensagens; e Ivo, o adversário. As mensagens passam por um canal de

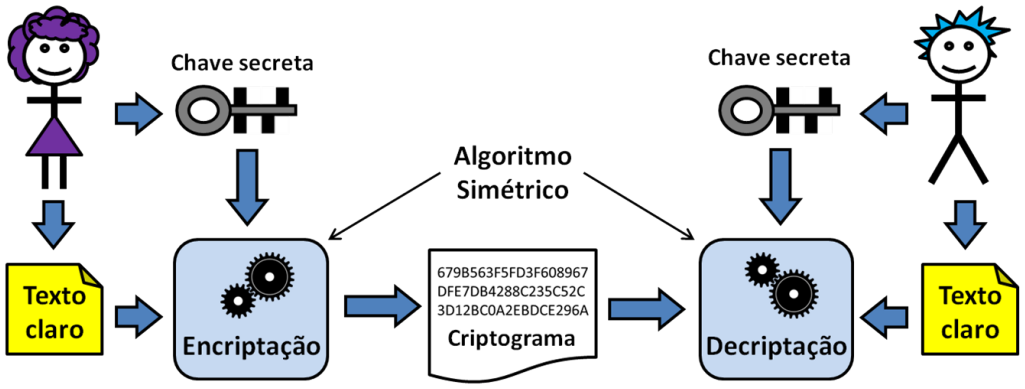


Figura 5.4. Sistema criptográfico simétrico (de [Braga and Dahab 2015]).

comunicação inseguro e controlado por Ivo. O algoritmo criptográfico é usado para transformar o texto claro (legível por qualquer um) em texto encriptado (o criptograma legível apenas por Ana e Beto) e vice-versa.

A chave criptográfica é o parâmetro de configuração do algoritmo que viabiliza a recuperação de um texto claro a partir do texto encriptado. Ana e Beto usam uma chave criptográfica conhecida apenas por eles e compartilhada (ou combinada) por um canal seguro diferenciado. A segurança do sistema criptográfico reside no segredo da chave e não no segredo do algoritmo criptográfico. Assim, se um algoritmo de boa reputação é usado, a qualidade da implementação deste algoritmo e o tamanho da chave criptográfica determinam a dificuldade em violar a encriptação da mensagem.

Existem dois tipos de sistemas criptográficos, comumente conhecidos como criptografia de chave secreta (ou simétrica) e criptografia de chave pública (ou assimétrica). Na criptografia de chave secreta, uma única chave é usada para encriptar e decriptar a informação. Na criptografia de chave pública, duas chaves são necessárias. Uma chave é usada para encriptar; a outra chave, diferente, é usada para decriptar a informação. O desempenho (a velocidade) da criptografia simétrica é geralmente superior a da criptografia assimétrica, no mesmo nível de segurança.

Por causa das questões administrativas de distribuição de chaves, a criptografia de chave pública é recomendada para grupos grandes e ambientes dinâmicos e abertos. Por outro lado, a criptografia de chave secreta é recomendada para grupos pequenos onde as partes comunicantes já estão bem definidas. Conforme descrito adiante no texto, uma solução amplamente utilizada pelos protocolos de comunicação segura na Internet usa uma combinação das tecnologias simétrica e assimétrica, chamada de sistemas criptográficos híbridos.

5.3.1. Criptografia simétrica (de chave secreta)

A Figura 5.4 ilustra os passos de encriptação e decriptação com algoritmos simétricos de chave secreta. Apenas a chave usada na encriptação pode decriptar corretamente a informação. Por isso, esta chave deve ser protegida e guardada em segredo; daí o nome de chave secreta. Este sistema criptográfico poderia ser diretamente utilizado para encriptação bidirecional com a mesma chave. Porém, na prática, diversos detalhes de implementação dificultam a utilização segura da mesma chave para encriptação nas duas direções do canal de comunicação.

Na criptografia simétrica, a chave secreta deve ser conhecida por todos aqueles que

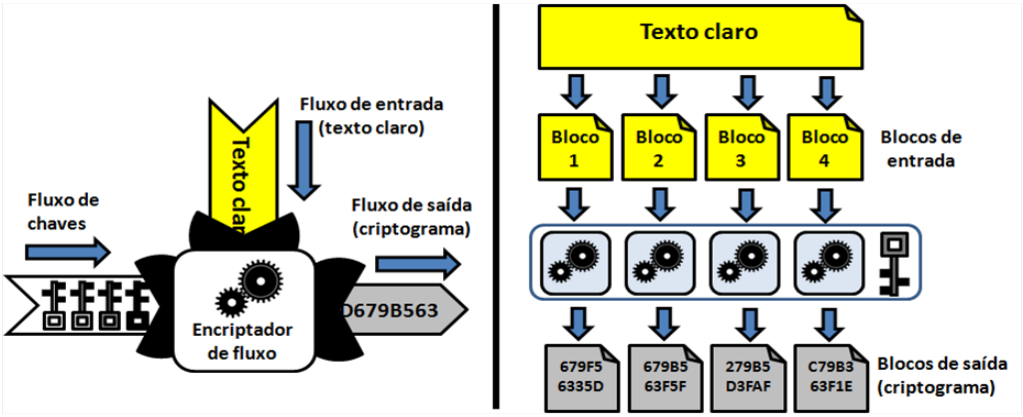


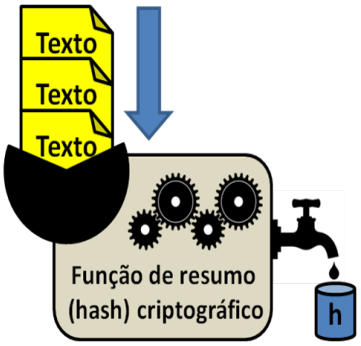
Figura 5.5. Encriptação simétrica de fluxo e de blocos (de [Braga and Dahab 2015]).

precisam decriptar a informação encriptada com ela. Um dos problemas práticos é o compartilhamento ou distribuição segura da chave secreta. A distribuição de chaves é um aspecto importante da criptografia de chave secreta. Apesar das dificuldades de distribuição de chaves, a criptografia de chave secreta é muito usada. Suas dificuldades aparentes podem ser contornadas pela combinação desta tecnologia com a criptografia de chave pública, originando sistemas criptográficos híbridos.

Existem duas categorias de algoritmos simétricos de encriptação (Figura 5.5): de fluxo e de bloco. Na encriptação de blocos, o texto claro é quebrado em blocos (de bits) de tamanho fixo. A encriptação ocorre sobre blocos individualmente e produz criptogramas em blocos também. O tamanho da chave criptográfica é geralmente um múltiplo do tamanho do bloco. A encriptação de fluxo trabalha sobre sequências (de bits). A sequência ou fluxo de entrada é transformado continuamente na sequência ou fluxo de saída, bit a bit. Neste caso, é importante que a chave criptográfica seja uma sequência de bits pelo menos do mesmo tamanho do fluxo de entrada. Na prática, os bits da chave podem ser produzidos por um gerador de sequências de bits pseudoaleatórias, a partir de uma chave mestra de tamanho fixo. Exemplos de algoritmos criptográficos simétricos comumente usados atualmente são o AES, o SERPENT, e o 3DES.

Os algoritmos simétricos de bloco trabalham sobre dados com tamanho igual a um múltiplo inteiro do tamanho do bloco. Já o texto claro tem tamanho livre. Para obter flexibilidade no texto claro, é necessário um mecanismo de preenchimento de blocos incompletos a fim de que o tamanho do texto claro seja múltiplo do tamanho do bloco. Chama-se *padding* (preenchimento) o completamento do texto claro para que ele seja múltiplo do tamanho do bloco do algoritmo de encriptação.

Os modos de operação da encriptação de blocos combinam de maneiras diferentes os blocos do texto claro (ou do criptograma) com a chave criptográfica, para produzir encadeamentos de blocos encriptados característicos de cada modo de operação. Assim, o mesmo texto claro, encriptado com a mesma chave, mas usando modos de operação diferentes, resulta em criptogramas diferentes. Os modos de operação mais comuns são ECB, CBC, CFB, OFB e CTR [NIST 2001]. Os modos ECB e CBC são típicos da encriptação de blocos. Os modos CFB, OFB e CTR se comportam como na encriptação de fluxo. Os modos de operação, exceto o ECB, evitam a ocorrência de padrões identificáveis nos bits do criptograma. Os modos de



Seja *H(x)* = *h*, onde *x* é o texto claro, *H*() é a função de resumo e *h* é o valor hash. As seguintes propriedades são verdadeiras:

- *H(x)* pode ser aplicada a um x de qualquer tamanho;
- *H(x)* produz uma saída h de tamanho fixo;
- É eficiente calcular *H(x)* para qualquer x;
- Para um *h* qualquer, é impossível calcular um *x*, tal que *H(x)* = *h*;
- Para um *x*, é impossível achar *y*, tal que *H(x)* = *H(y)*;
- É impossível achar um par qualquer *(x,y)*, tal que *H(x)* = *H(y)*.

Figura 5.6. Funções de resumo (hash) criptográfico (de [Braga and Dahab 2015]).

operação são diferentes em relação à recuperação de erros e à perda de sincronismo.

5.3.2. Resumo criptográfico (*hash*) e Código de Autenticação de Mensagem (MAC)

O uso de encriptação somente não é capaz de garantir que o criptograma não foi corrompido, maliciosamente modificado, ou até mesmo completamente substituído por Ivo, quando em trânsito de Ana até Beto. Em uma solução incial simples e incompleta desta situação, um valor de *hash* calculado sobre o criptograma poderia ser enviado por Ana junto com o criptograma e verificado por Beto. Deste modo, as corrupções acidentais ou maliciosas do criptograma poderiam ser detectadas por Beto.

Uma função de resumo criptográfico, também chamada de função de *hash* segura, gera uma sequência de bits, chamado de valor do *hash*, único para os dados de entrada da função. O *hash* é muito menor que o texto claro de entrada e geralmente tem um tamanho fixo de dezenas (algumas centenas) de bits. Em princípio, a função de *hash* é unidirecional porque não é reversível; isto é, não é computacionalmente possível recuperar o texto claro original a partir da sequência binária do *hash*. Além disso, idealmente, não é fácil achar dois conjuntos de dados (dois textos claros) que geram o mesmo valor de *hash*. A Figura 5.6 ilustra o comportamento das funções de *hash* seguras e também descreve algumas das propriedades técnicas destas funções que as tornam importantes para a segurança criptográfica.

As funções de resumo criptográfico não resolvem todos os problemas de integridade da mensagem. Por exemplo, mesmo os criptogramas acompanhados de valores *hash* ainda podem ser substituídos (incluindo a troca do *hash*) quando da transmissão de uma mensagem por um canal inseguro. Este problema é resolvido pelo uso de uma *tag* de autenticação no lugar do *hash*. As funções de resumo criptográfico podem ser usadas como blocos de construção das funções que calculam os códigos de autenticação de mensagens (*Message Authentication Codes* - MAC) baseados em *hash* e chamados de HMAC. Em termos simples, um HMAC é um *hash* chaveado e pode ser usado como um mecanismo de autenticação simples quando as partes comunicantes compartilham uma chave secreta. Neste caso, diz-se que a mensagem é autêntica para Ana e Beto apenas, mas não para um terceiro, uma vez que o terceiro não consegue determinar com certeza, quem (Ana ou Beto) gerou o MAC da mensagem.

Um código de autenticação de mensagem (MAC) é um mecanismo criptográfico que produz um selo (*tag*) de autenticidade baseado em uma chave compartilhada entre Ana e Beto e, por isto, relativa apenas a eles. A função MAC recebe como entrada a chave simétrica e

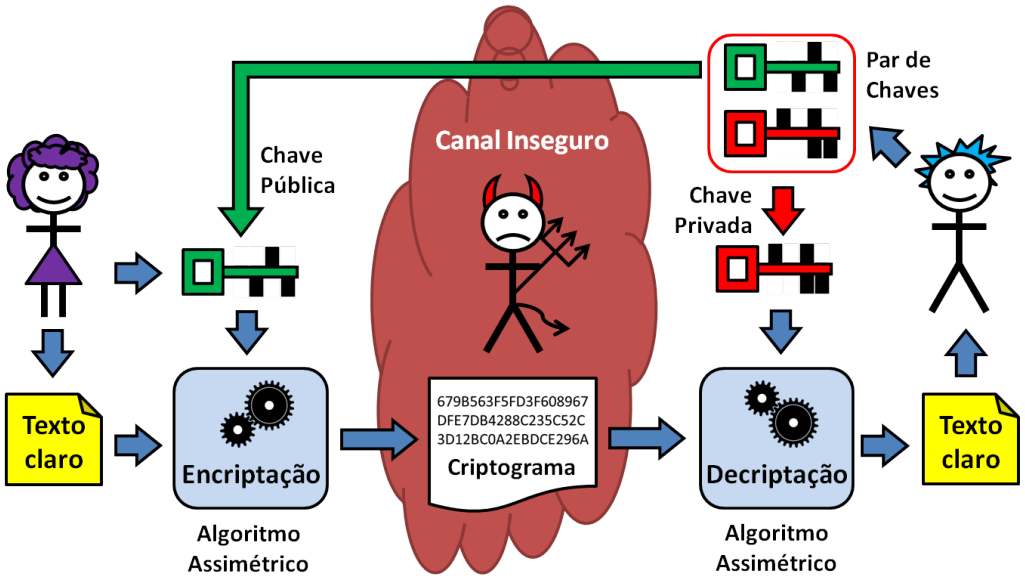


Figura 5.7. Sistema criptográfico assimétrico para sigilo (de [Braga and Dahab 2015]).

a mensagem (texto claro) e, utilizando em seu algoritmo funções de encriptação ou de *hash*, produz a *tag*. A verificação da *tag* é feita pela comparação da *tag* recebida com uma nova *tag* calculada no recebimento da mensagem.

A encriptação autenticada combina em uma única função criptográfica as funções de encriptação e de MAC. A encriptação autenticada é uma função criptográfica relativamente nova que atende aos dois objetivos (encriptação e autenticação) simultaneamente, sendo geralmente materializada por modos de operação específicos para algoritmos de encriptação de blocos, tais como os modos de operação GCM [NIST 2007] do AES e CCM, aplicável a qualquer encriptação de blocos.

5.3.3. Criptografia assimétrica (de chave pública)

Na criptografia de chave pública, uma das chaves do par é dita privada (a de decriptação), a outra é feita pública (a de encriptação). Estas duas chaves são matematicamente relacionadas e trabalham aos pares, de modo que o criptograma gerado com uma chave deve ser decriptado pela outra chave do par. Nenhuma das chaves do par pode ser usada sozinha em um sistema criptográfico assimétrico completo. A criptografia assimétrica pode ser usada tanto para encriptação (atuando na preservação de sigilo), quanto para assinaturas digitais (atuando na garantia de autenticidade e de irrefutabilidade). A criptografia de chave pública é indispensável para o sigilo e a privacidade na Internet, tornando possível a comunicação privada em redes públicas.

5.3.3.1. Encriptação assimétrica para sigilo

Na garantia de sigilo, a encriptação com a chave pública torna possível que qualquer um envie criptogramas (textos cifrados) para o dono da chave privada. A Figura 5.7 ilustra, com os mesmo personagens anteriores, um sistema criptográfico assimétrico para sigilo e seus elementos básicos. As mensagens de Ana para Beto são transmitidas por um canal inseguro, controlado

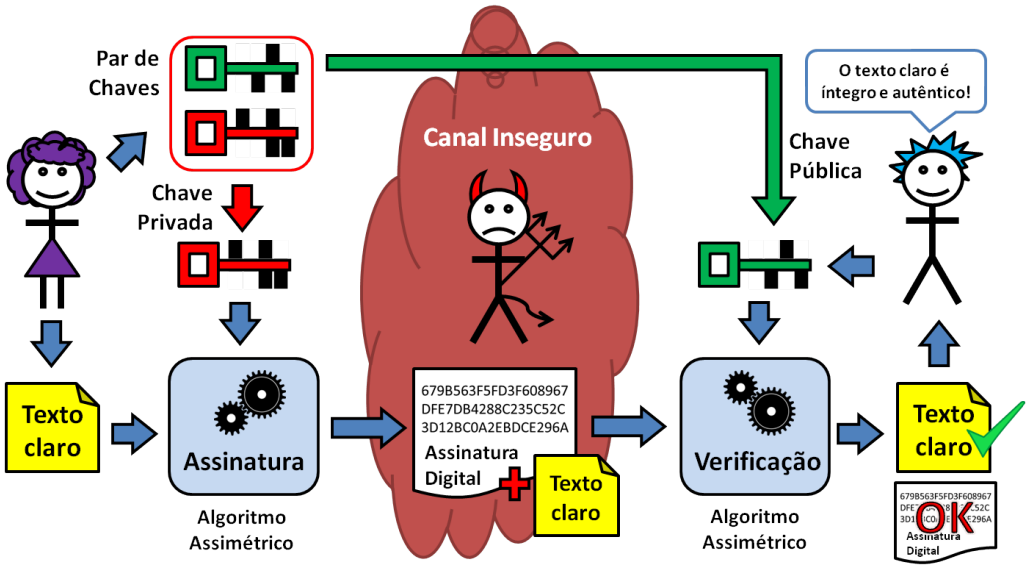


Figura 5.8. Criptografia assimétrica para autenticidade (de [Braga and Dahab 2015]).

por Ivo. Beto possui um par de chaves, uma chave pública e outra privada. Ana conhece a chave pública de Beto, mas somente o dono do par de chaves (Beto) conhece a chave privada.

Na Figura 5.7, observa-se que Ana envia uma mensagem privada para Beto. Para fazer isso, Ana encripta a mensagem usando a chave pública de Beto. Ana conhece a chave pública de Beto, que foi divulgada por ele previamente. Porém, o criptograma só pode ser decriptado pela chave privada de Beto; nem Ana pode fazê-lo. Para obter comunicação segura bidirecional, acrescenta-se ao sistema criptográfico o mesmo processo no sentido oposto (de Beto para Ana), com outro par de chaves. Isto é, Beto usa a chave pública de Ana para enviar mensagens encriptadas para ela. Já Ana usa sua própria chave privada pessoal para decriptar a mensagem enviada por Beto.

5.3.3.2. Criptografia assimétrica para autenticidade e irrefutabilidade

A criptografia de chave pública é a base para outros dois serviços criptográficos importantes para a proteção das comunicações: a autenticação das partes comunicantes e a verificação de integridade das mensagens. Os passos de operação da criptografia de chave pública para autenticação de mensagens, em certa medida, são o oposto do uso para sigilo. A criptografia de chave pública para assinatura digital é usada para obter integridade, autenticidade e irrefutabilidade.

Chama-se assinatura digital ao resultado de uma certa operação criptográfica com a chave privada sobre o texto claro. Neste caso, o dono da chave privada pode gerar mensagens assinadas, que podem ser verificadas por qualquer um que conheça a chave pública correspondente, portanto, sendo capaz de verificar a autenticidade da assinatura digital. Do ponto de vista das implementações criptográficas subjacentes, nem sempre a operação de assinatura é uma encriptação e nem a sua verificação é sempre uma decriptação.

Visto que qualquer um de posse da chave pública pode *"decriptar"* a assinatura digital, ela não é mais secreta, mas possui outra propriedade: a irrefutabilidade. Isto é, quem quer que

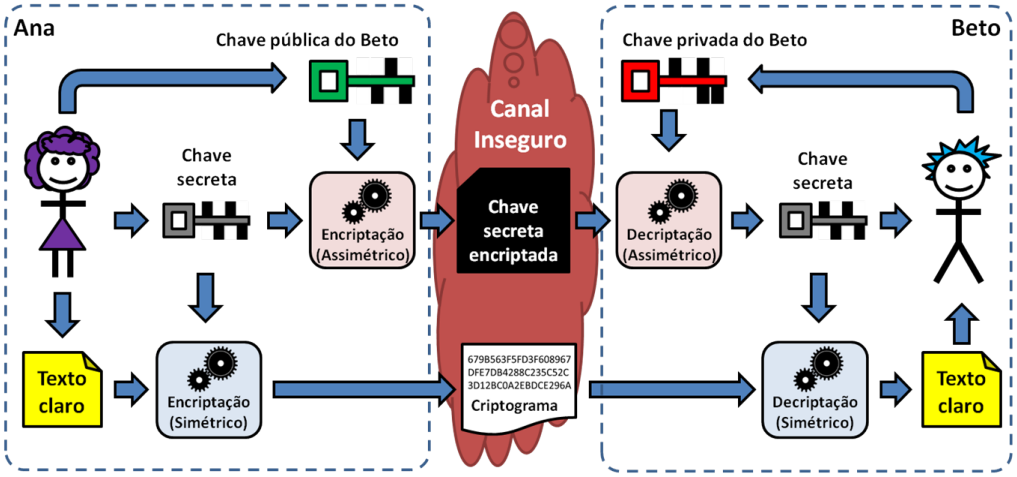


Figura 5.9. Transporte de chave de sessão (de [Braga and Dahab 2015]).

verifique a assinatura com a chave pública, sabe que ela foi produzida por uma chave privada exclusiva; logo, a mensagem não pode ter sido gerada por mais ninguém além do proprietário da chave privada. Na Figura 5.8, Ana usa sua chave privada para assinar digitalmente uma mensagem para Beto. O texto claro e a assinatura digital são enviados por um canal inseguro (sem sigilo) e podem ser lidos por todos, por isso a mensagem não é secreta. Qualquer um que conheça a chave pública de Ana (todo mundo, inclusive Beto), pode verificar a assinatura digital. Ivo pode ler a mensagem, mas não consegue falsificá-la de maneira indetectável, pois não conhece a chave privada de Ana.

5.3.4. Transporte de chaves de sessão e sistemas criptográficos híbridos

Há ocasiões em que entidades que nunca antes compartilharam chaves criptográficas precisam se comunicar em sigilo. Nestes casos, uma chave efêmera, usada apenas para algumas encriptações decorrentes de uma conversa, pode ser gerada por uma das partes e transportada com segurança para a outra parte, momentos antes do início da conversa. Tal transporte seguro de chaves pode ser implementado por um sistema criptográfico híbrido e emprega o conceito de chave de sessão, em que uma chave secreta é encriptada por uma chave pública para transporte seguro. Assim, a criptografia assimétrica (tipicamente, o RSA) é usada como canal seguro para o compartilhamento da uma chave secreta (por exemplo, do AES) usada na encriptação de dados da comunicação.

A Figura 5.9 mostra um sistema criptográfico híbrido usado para transporte de uma chave de sessão. Durante uma comunicação segura, a encriptação assimétrica só é executada uma única vez, no início da comunicação, para compartilhar a chave secreta. A partir do segundo criptograma da conversa, basta para Ana encriptar com a chave de sessão. A tarefa de Beto é recuperar a chave secreta e manter a comunicação com Ana usando a chave secreta de sessão para proteger os dados. De modo análogo, na decriptação, a recuperação da chave secreta só ocorre no início da comunicação. Observados os detalhes de implementação (por exemplo, derivando-se duas chaves de sessão a partir do segredo compartilhado), depois que ambos (Ana e Beto) possuem a chave secreta da sessão, a comunicação pode ocorrer nos dois sentidos. Isto é, tanto de Ana para Beto, quanto de Beto para Ana.

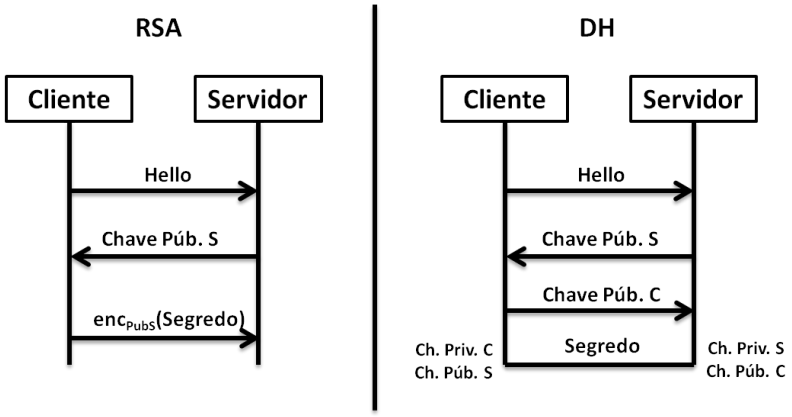


Figura 5.10. Transporte de chave com RSA e acordo de chave com DH.

5.3.5. RSA e Diffie-Hellman (DH)

Tradicionalmente, o algoritmo criptográfico assimétrico mais conhecido para encriptação e transporte de chaves é o RSA (publicado em 1978 [Rivest et al. 1978]), cujo nome é formado pelas letras iniciais dos sobrenomes dos seus autores: Ron Rivest, Adi Shamir e Leonard Adleman. Na prática, a versão acadêmica original (canônica) nunca deve ser utilizada. Implementações padronizadas que adotam mecanismos de preenchimento e aleatorização considerados seguros devem ser preferidas em vez das implementações fora dos padrões.

Para promover segurança e interoperabilidade, o uso e a implementação do RSA devem obedecer aos padrões internacionais. O documento PKCS#1v2 [Jonsson and Burt Kaliski 2003] especifica o *Optimal Asymmetric Encryption Padding* (OAEP) como um mecanismo de preenchimento (*padding*), que transforma o RSA em um mecanismo de encriptação assimétrica aleatorizado chamado RSA-OAEP. Já o *Probabilistic Signature Scheme* (PSS) é um esquema de assinaturas digitais probabilísticas com RSA (RSA-PSS) também padronizado no PKCS#1v2 [Jonsson and Burt Kaliski 2003] para substituir o esquema de assinatura do PKCS#1v1, considerado inseguro.

Outra maneira de resolver a questão do compartilhamento de chaves criptográficas entre entidade que nunca se encontraram, mas que precisam se comunicar em sigilo, é por meio dos protocolos de acordo de chaves. Os métodos de acordo de chaves são utilizados para combinar ou negociar uma chave secreta entre dois ou mais participantes usando um canal público. Uma característica interessante destes métodos é que o segredo compartilhado (a partir do qual a chave será derivada) é combinado pela troca de informações públicas por meio de um canal inseguro.

O protocolo de acordo de chaves Diffie-Hellman (DH), publicado em 1976 no artigo que lançou a criptografia de chave pública [Diffie and Hellman 1976], é o método mais usado na Internet para o acordo de um segredo compartilhado entre duas partes. Assim como no caso do RSA, as implementações canônicas do DH não devem ser usadas por que elas são vulneráveis aos ataques de personificação de uma ou ambas as partes (o ataque *Man-In-The-Middle* - MITM). Duas variações do DH original são aquela com parâmetros efêmeros (DHE) e aquela com autenticação das partes (DHA). Também há variações do DH implementadas como a criptografia de curvas elípticas. Figura 5.10 ilustra as diferenças entre as duas abordagens de distribuição de chaves, RSA e DH, quando usadas em protocolos como o SSL/TLS.

5.3.6. Criptografia de Curvas Elípticas

Os sistemas criptográficos baseados em curvas elípticas foram propostos por volta de 1985 de forma independente por dois pesquisadores [Miller 1985, Koblitz 1987], e se baseiam na dificuldade de se calcular o logaritmo discreto no grupo de pontos induzido por uma curva elíptica definida sobre um corpo finito F_{p^m}, onde p é um número primo e m é um inteiro positivo. Neste texto nos limitamos aos seguintes casos: $p \neq 2, m = 1$, isto é, corpos *primos*; e $p = 2, m \geq 1$, chamados de corpos *binários*. A forma geral da equação de uma curva elíptica é $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ [Hankerson et al. 2004]. Para corpos primos simplifica-se a forma da curva: $y^2 = x^3 + ax + b$, com a,b em F_p. Para corpos binários, a forma da curva é $y^2 + xy = x^3 + ax^2 + b$, com a,b em F_{2^m}. O tamanho das chaves criptográficas nos criptossistemas de curvas elípticas é consideravelmente menor que nos sistemas criptográficos assimétricos tradicionais, como o RSA ou ElGamal. A criptografia de curvas elípticas é usada em trocas de chaves, assinaturas digitais e encriptação. Os esquemas e protocolos criptográficos sobre curvas elípticas mais comumente utilizados atualmente são o protocolo de acordo de chaves *Elliptic Curve Diffie Hellman* (ECDH), o esquema de assinaturas digitais *Elliptic Curve Digital Signature Algorithm* (ECDSA) [NIST 2013] e a encriptação *Elliptic Curve Integrated Encryption Scheme* (ECIES) [Hankerson et al. 2004].

O NIST [NIST 2013] recomenda cinco curvas elípticas sobre corpos primos para serem utilizadas no ECDSA, em cinco níveis de segurança, a saber: P-192, P-224, P-256, P-384 e P-521 (também conhecidas [SEC 2010] como secp192r1, secp224r1, secp256r1, secp384r1, secp521r1). Há uma variedade de curvas definidas por diversas instituições, muitas já consideradas inseguras para os padrões de segurança atuais. Há também padrões emergentes, como a curva 25519 [Bernstein 2006], que tem sido considerada um novo padrão de fato na indústria de segurança criptográfica e uma substituta para as curvas padronizadas pelo NIST [NIST 2013]. Esta curva é mais rápida, possivelmente mais segura, e alegadamente sem a influência de agências de governo sobre seu projeto (conforme `https://cr.yp.to/ecdh.html`).

5.3.7. Tamanhos de chaves criptográficas e níveis de segurança

Sistemas criptográficos implementados em software geralmente combinam diversas funções criptográficas, cujas configurações devem ser escolhidas de modo a manter as funções no mesmo nível de segurança. A Tabela 5.1 (apresentada em [Braga and Dahab 2018]) mostra uma comparação entre os níveis de segurança de tipos de primitivas criptográficas e os tamanhos de chaves correspondentes. O nível de segurança é uma medida relativa e pode ser interpretado como sendo a força de um algoritmo criptográfico simétrico com chave de n bits.

A tabela foi adaptada de `keylength.com` e considera as recomendações do NIST (2016) [BlueKrypt]. Cada linha da tabela representa um nível de segurança, de 80 bits até 256 bits. Por exemplo, no nível de 80 bits, atualmente útil apenas para sistemas legados, chaves RSA de 1024 bits são comparáveis às chaves DH de 1024 bits com segredo de 160 bits, uma curva elíptica de 160 bits e *hashes* de 160 bits também. Já no nível de segurança mais alto, 256 bits, o RSA se torna inviável na prática com chaves de 15360 bits, enquanto as curvas elípticas despontam com chaves de 512 bits e *hashes* de 512 bits para assinaturas e de até 384 bits para geração de chaves.

Tabela 5.1. Níveis de segurança e tamanhos de chave (de [BlueKrypt]).

Nível de segurança	Fatoração (IFP)	DLP		Curva Elíptica	Hash	
		chave	Corpo		Assinatura	KDF/HMAC/PRNG
80	1024	160	1024	160–163	SHA1 (160)	–
112	2048	224	2048	224–233	SHA-224/512 SHA3-224	–
128	3072	256	3072	256–283	SHA-256/512 SHA3-256	SHA1(160)
192	7680	384	7680	384–409	SHA-384 SHA3-384	SHA-224 SHA-512
256	15360	512	15360	512–571	SHA-512 SHA3-512	SHA-256/384/512 SHA-512/SHA3-512

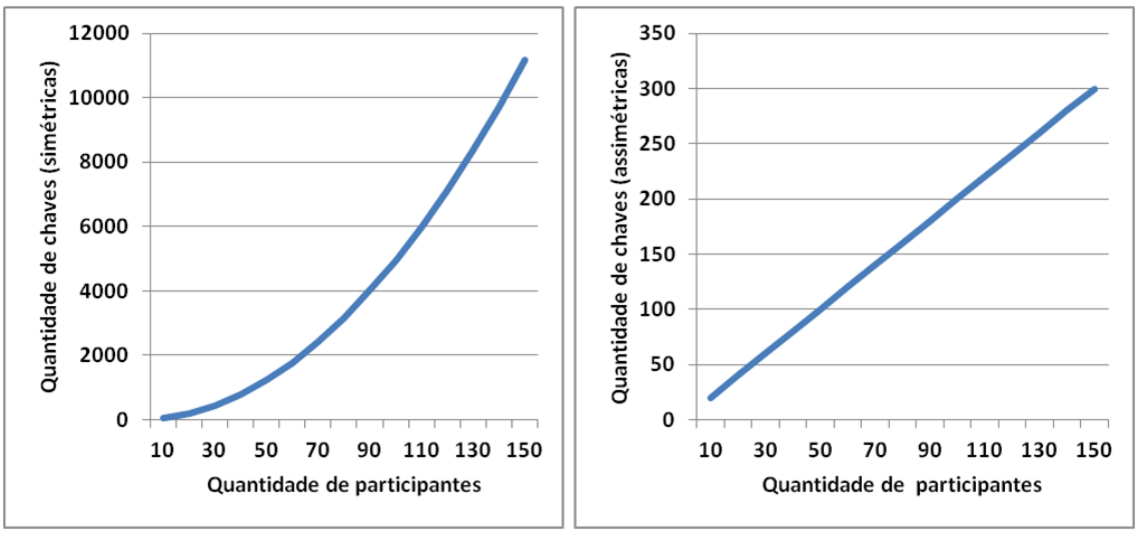


Figura 5.11. Quantidades de chaves assimétricas e simétricas para 150 participantes (de [Braga and Dahab 2018]).

5.3.8. Distribuição de chaves públicas com certificação digital

A criptografia assimétrica (de chaves públicas) tem a propriedade de simplificar o problema de distribuição de chaves criptográficas. Por exemplo, cada um dos participantes do sistema criptográfico só precisa ter o seu par de chaves, manter em segredo a sua chave privada e divulgar a chave pública. A Figura 5.11 ilustra a distribuição de chaves públicas para um grupo de quatro indivíduos. O gráfico da direita na Figura 5.11 mostra o crescimento da quantidade de chaves (públicas e privadas) como uma função afim da quantidade de participantes, onde quantidade de chaves é o dobro da quantidade de participantes. Já a distribuição de chaves simétricas (gráfico da esquerda) obedece a uma função quadrática.

O principal problema administrativo associado à distribuição de chaves públicas é justamente a confiança depositada na chave distribuída publicamente. Se a chave pública de alguém pode ser facilmente encontrada em qualquer lugar, então é difícil assegurar com alto grau de certeza que esta chave não foi corrompida ou substituída. O problema de garantir a integridade e a autenticidade da chave pública é muito importante e, se não for solucionado satisfatoriamente, pode comprometer a confiança no sistema criptográfico inteiro. Uma maneira de validar chaves públicas é fazer com que elas sejam emitidas por Autoridades Certificadoras

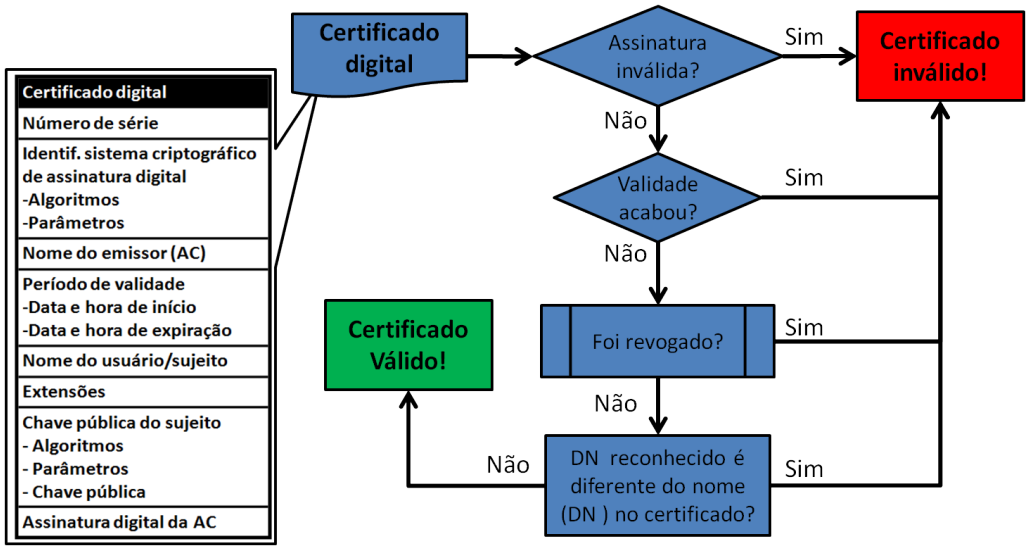


Figura 5.12. Fluxograma de validação de um certificado digital (de [Braga and Dahab 2015]).

(AC) de uma Infraestrutura de Chaves Públicas (ICP), do inglês *Public-Key Infrastrucuture* - PKI, que torne possível a verificação da autenticidade de tais chaves.

Certificação digital e os aspectos de validação de certificados digitais possuem atualmente boas práticas bem documentadas [NIST 2012] e literatura especializada na implantação de tais infraestruturas [Chandra et al. 2002]. Um certificado digital de chave pública, ou simplesmente certificado, é uma estrutura de dados que dá como verdadeiro o vínculo entre uma chave pública autêntica e uma entidade cujo nome está no certificado. A veracidade do certificado é garantida por uma terceira parte confiável, emissora do certificado, chamada de Autoridade Certificadora (CA). O certificado contém a assinatura digital da CA emissora, a chave pública e a identidade da entidade reconhecida pela CA, além de outras informações, tais como as datas de início e de final de validade e o uso pretendido da chave, entre outras. Por isto, o certificado é usado na verificação de que uma chave pública válida pertence a uma entidade e serve também como meio confiável para distribuição de chaves públicas.

A validação do certificado é realizada toda vez que a confiança na autenticidade da chave pública contida nele deve ser garantida. A assinatura da AC pode ser verificada por qualquer um com acesso à chave pública da AC, cujo certificado é amplamente disponível. Por exemplo, num caso bastante comum, uma AC pode emitir certificados SSL/TLS para servidores web que se comunicam por meio do protocolo HTTP sobre TLS (HTTPS). Quando um software cliente HTTPS (o navegador ou *browser*) faz uma requisição SSL/TLS de saudação para um servidor web protegido, em que a resposta do servidor inclui o seu certificado digital. O software cliente HTTPS valida o certificado do servidor verificando a assinatura da AC emissora sobre a chave pública do servidor e outros parâmetros do certificado. Se o cliente já não possuir a chave pública da AC, ele vai buscá-la (em um repositório de chaves públicas da AC). Se o certificado é verificado como válido, então o software cliente acredita que o servidor é autêntico.

A validação do certificado (e da chave pública contida nele) abrange mais etapas do que somente a verificação da assinatura da AC. Além da verificação da assinatura, o software cliente precisa verificar se o certificado não atingiu o final de seu período de validade, se não foi

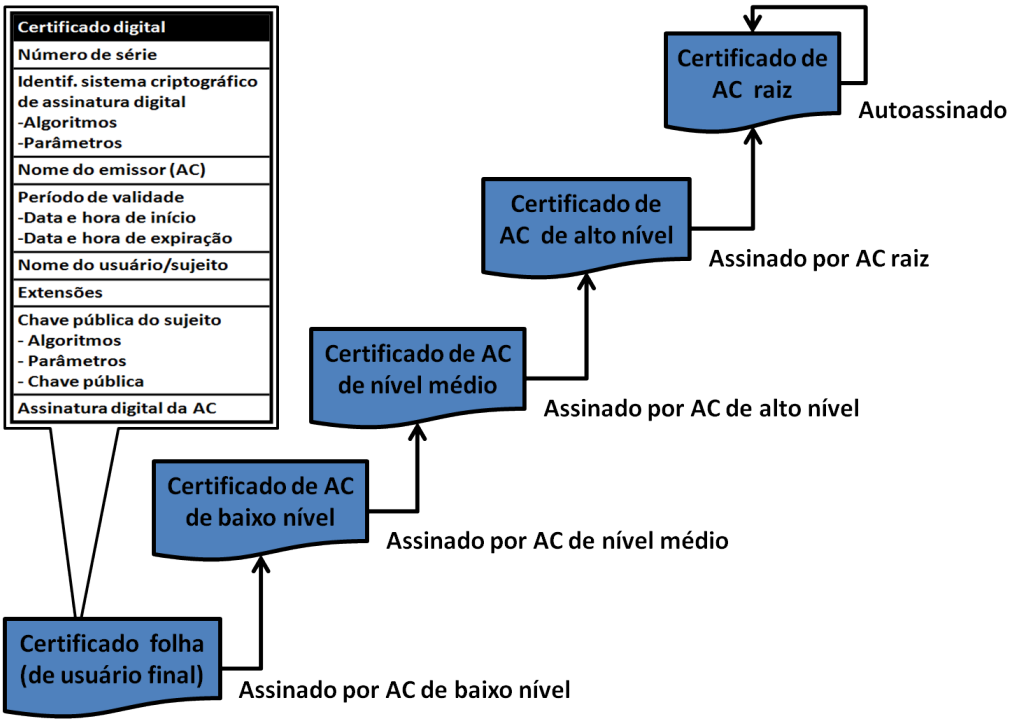


Figura 5.13. Cadeia hierárquica de certificação digital (de [Braga and Dahab 2015]).

revogado e se o nome constante no certificado é o mesmo da entidade que alega ser a dona da chave pública. O nome também deve ser obtido de um terceiro confiável, por exemplo, de um serviço de DNS seguro, no caso de nomes de domínio. A validação do certificado é ilustrada no fluxograma da Figura 5.12.

A verificação da assinatura da AC em um certificado digital exige a chave pública da AC. Para ser confiável, a chave pública da AC deve estar contida em um certificado assinado por outra AC ou autoassinado, se for uma CA raiz. As verificações sucessivas de uma sequência de assinaturas formam uma cadeia de certificação ou hierarquia de certificados. Os certificados na base da hierarquia são assinados pelas ACs de mais baixo nível, cujos certificados são assinados pelas ACs intermediárias, que têm seus certificados assinados pelas ACs de alto nível, cujos certificados são assinados pela AC raiz, que tem seus certificados autoassinados. A Figura 5.13 ilustra esta cadeia de certificação.

Uma AC revoga um certificado nas seguintes situações: (*i*) quando ocorrem erros na emissão do certificado (por exemplo, quando o nome da entidade está escrito de forma errada); (*ii*) o certificado foi emitido para uso de um serviço específico que o portador não tem mais acesso (por exemplo, no caso de demissão de um funcionário); (*iii*) a chave privada do usuário final é comprometida (por exemplo, um *smartcard* foi perdido); ou ainda (*iv*) quando a chave privada da AC é comprometida (por exemplo, em uma situação extrema, quando a credencial de um administrador da AC é comprometida). Uma Lista de Certificados Revogados (*Certificate Revocation List* - CRL) é o documento assinado digitalmente pela AC que lista o número de série de todos os certificados, ainda não expirados, que perderam a utilidade por algum dos motivos acima. Um software criptográfico pode consultar um serviço de CRL (oferecido pela AC) para receber atualizações periódicas, em intervalos regulares definidos por procedimentos da AC.

5.4. Criptografia aplicada com `OpenSSL`

Além de ser uma ferramenta para utilização e testes do protocolo SSL/TLS, o software `OpenSSL` [OpenSSL.org] também é um biblioteca criptográfica completa com muitas funções de criptografia disponíveis por meio da sua interface de comandos de linha (*Command Line Interface* - CLI). Por isto, esta seção usa a CLI do `OpenSSL` para mostrar na prática os conceitos descritos na seção anterior. O `OpenSSL` é amplamente disponível, já que, por exemplo, faz parte das distribuições Linux. Já os usuários Windows devem baixar os binários do `OpenSSL` de uma fonte confiável, facilmente encontrada. Em qualquer caso, é importante verificar qual versão do `OpenSSL` está em uso.

Os exemplos mostrados a seguir neste capítulo foram executados com o `OpenSSL` para Windows em uma janela de terminal (do tipo DOS) e funcionam da mesma forma na versão para Linux. O leitor interessado em reproduzir as tarefas deste tutorial, quando pertinente, pode executar os comandos *OpenSSL* na sequência em que são apresentados nas listagens a seguir.

5.4.1. Identificação da versão instalada do `OpenSSL`

A Listagem 5.1 mostra dois modos de verificar a versão do `OpenSSL` instalado. A primeira é com o comando `openssl version`, da linha 1, que mostra o número de versão e a data de lançamento da versão. A segunda é com a opção `version -a` (linha 4) que lista todas as informações detalhadas sobre a versão. Como pode ser observado na Listagem 5.1 (linhas de 5 até 7), a versão 1.1.1 de agosto de 2018, para Windows de 32 bits, foi usada durante a escrita deste texto. Esta distribuição do `OpenSSL` já suporta a versão 1.3 do SSL/TLS.

Nas linhas de 7 até 17, a Listagem 5.1 mostra diversas informações relacionadas ao binário da distribuição do `OpenSSL` e seu processo de compilação e *build*. Estes são detalhes de implementação que não afetam o tutorial desta seção.

A Listagem 5.1 serve de modelo para as outras listagens mostradas ao longo do restante do texto. Por isto, o leitor deve se habituar a este formato de listagem de comandos em que uma sequência de comandos e suas saídas são mostradas na mesma listagem de console (janela de terminal) do sistema operacional.

Listagem 5.1. Descobrindo a versão do `OpenSSL`.

```
1 C:\>openssl version
2 OpenSSL 1.1.1  11 Sep 2018
3
4 C:\>openssl version -a
5 OpenSSL 1.1.1  11 Sep 2018
6 built on: Thu Sep 13 14:54:36 2018 UTC
7 platform: VC-WIN32
8 options:  bn(64,32) rc4(8x,mmx) des(long) idea(int) blowfish(ptr)
9 compiler: cl /Z7 /Fdossl_static.pdb /Gs0 /GF /Gy /MD /W3 /wd4090 /nologo /O2 /WX
10  -DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_BN_ASM_PART_WORDS -DOPEN
11 SSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_AS
12 M -DSHA512_ASM -DRC4_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM -DVPAES_ASM -DWHIRLPOO
13 L_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -DPADLOCK_ASM -DPOLY1305_ASM -D_USE_32BIT_T
14 IME_T -D_USING_V110_SDK71_ -D_WINSOCK_DEPRECATED_NO_WARNINGS
15 OPENSSLDIR: "C:\Program Files (x86)\Common Files\SSL"
16 ENGINESDIR: "C:\Program Files (x86)\OpenSSL\lib\engines-1_1"
17 Seeding source: os-specific
```

5.4.2. Os comandos principais da CLI de serviços criptográficos

O `OpenSSL` é uma ferramenta bastante flexível que pode, inclusive, ser utilizado como um provedor de serviços criptográficos com uma interface de linha da comando. A Listagem 5.2 contém a saída do comando `openssl help` que mostra os comandos padrão (nas linhas de 2 a 14), com enfase em dois comandos em particular.

Primeiro é o comando `dgst`, que é usado para cálculo de *hashes*, MACs e assinaturas digitais. O segundo é o comando `enc`, usado para encriptação simétrica. A Listagem 5.2, nas linhas de 16 até 22, mostra todas as funções de *hash* disponíveis na instalação do `OpenSSL`.

Em seguida, as linhas de 24 até 45 mostram as funções de encriptação simétrica no formato AAA-KKK-OOO, onde AAA é o algoritmo de encriptação, KKK é o tamanho de chave criptográfica e OOO é o modo de operação do algoritmo de encriptação simétrico. Cada comando tem seu próprio `help`, que deve ser explorado pelo leitor.

Listagem 5.2. Descobrindo os comandos do `OpenSSL`.

```
1 C:\>openssl help
2 Standard commands
3 asn1parse         ca              ciphers         cms
4 crl               crl2pkcs7       dgst            dhparam
5 dsa               dsaparam        ec              ecparam
6 enc               engine          errstr          gendsa
7 genpkey           genrsa          help            list
8 nseq              ocsp            passwd          pkcs12
9 pkcs7             pkcs8           pkey            pkeyparam
10 pkeyutl          prime           rand            rehash
11 req              rsa             rsautl          s_client
12 s_server         s_time          sess_id         smime
13 speed            spkac           srp             storeutl
14 ts               verify          version         x509
15
16 Message Digest commands (see the 'dgst' command for more details)
17 blake2b512       blake2s256      gost            md4
18 md5              mdc2            rmd160          sha1
19 sha224           sha256          sha3-224        sha3-256
20 sha3-384         sha3-512        sha384          sha512
21 sha512-224       sha512-256      shake128        shake256
22 sm3
23
24 Cipher commands (see the 'enc' command for more details)
25 aes-128-cbc      aes-128-ecb     aes-192-cbc     aes-192-ecb
26 aes-256-cbc      aes-256-ecb     aria-128-cbc    aria-128-cfb
27 aria-128-cfb1    aria-128-cfb8   aria-128-ctr    aria-128-ecb
28 aria-128-ofb     aria-192-cbc    aria-192-cfb    aria-192-cfb1
29 aria-192-cfb8    aria-192-ctr    aria-192-ecb    aria-192-ofb
30 aria-256-cbc     aria-256-cfb    aria-256-cfb1   aria-256-cfb8
31 aria-256-ctr     aria-256-ecb    aria-256-ofb    base64
32 bf               bf-cbc          bf-cfb          bf-ecb
33 bf-ofb           camellia-128-cbc camellia-128-ecb camellia-192-cbc
34 camellia-192-ecb camellia-256-cbc camellia-256-ecb cast
35 cast-cbc         cast5-cbc       cast5-cfb       cast5-ecb
36 cast5-ofb        des             des-cbc         des-cfb
37 des-ecb          des-ede         des-ede-cbc     des-ede-cfb
38 des-ede-ofb      des-ede3        des-ede3-cbc    des-ede3-cfb
39 des-ede3-ofb     des-ofb         des3            desx
40 idea             idea-cbc        idea-cfb        idea-ecb
41 idea-ofb         rc2             rc2-40-cbc      rc2-64-cbc
42 rc2-cbc          rc2-cfb         rc2-ecb         rc2-ofb
43 rc4              rc4-40          seed            seed-cbc
44 seed-cfb         seed-ecb        seed-ofb        sm4-cbc
45 sm4-cfb          sm4-ctr         sm4-ecb         sm4-ofb
```

5.4.3. Codificação Base64 e geração de números pseudo-aleatórios

Algoritmos criptográficos produzem resultados em sequências binárias que, muitas vezes, são transmitidos ou armazenados em formato texto. Para isto, o `OpenSSL` oferece o comando `enc -base64` para codificação de um arquivo de entrada qualquer no formato Base64, aplicando uma regra simples: cada 3 bytes (24 bits) binários viram 4 caracteres imprimíveis (cada um com 6 bits e completados para um byte). A codificação Base64 não é uma função criptográfica e não tem finalidade de segurança. A Listagem 5.3 mostra como codificar e decodificar no formato Base64 um arquivo `gabarito.txt`. Este arquivo é usado em vários exemplos neste capítulo e (como seu nome sugere) simula o gabarito de uma prova, uma informação que certamente deve ser protegida de várias formas diferentes.

Listagem 5.3. Codificação em Base64 com `OpenSSL`.

```
1 C:\>type gabarito.txt
2 Gabarito da prova
3 1-a 2-c 3-d 4-c 5-a 6-b 7-d 8-d 9-a 10-c
4
5 C:\>openssl enc -base64 -in gabarito.txt -out gabarito.b64
6
7 C:\>type gabarito.b64
8 R2FiYXJpdG8gZGEgcHJvdmENCjEtYSAyLWMgMy1kIDQtYyA1LWEgNi1iIDctZCA4LWQgOS1hIDEwLWMNCg==
9
10 C:\>openssl enc -d -base64 -in gabarito.b64 -out gabarito2.txt
11
12 C:\>type gabarito2.txt
13 Gabarito da prova
14 1-a 2-c 3-d 4-c 5-a 6-b 7-d 8-d 9-a 10-c
```

Outro comando do `OpenSSL` bastante útil é o `rand` usado na geração de sequências de bytes pseudo-aleatórias. A Listagem 5.4 mostra diversas maneiras de utilizar o comando `rand`. A linha 1 mostra como gerar uma sequência pseudo-aleatória em hexadecimal de 20 bytes. As linhas 4 e 7 geram sequências pseudo-aleatórias de 32 e 64 bytes, respectivamente. A linha 11 formata a sequência de saída em base64. A linha 14 escreve a sequência de saída em um arquivo com a opção `-rand`. Isto é útil para fornecer uma fonte de aleatoriedade para outras funções criptográficas, como por exemplo a geração de chaves. A linha 16 mostra o conteúdo do arquivo. A linha 19 grava o arquivo e mostra a saída no formato Base64.

Listagem 5.4. Geração de números pseudo-aleatórios.

```
1 C:\>openssl rand -hex 20
2 031954f853219d94295a4dad7feefd85bbd5096e
3
4 C:\>openssl rand -hex 32
5 9047b1cc5050be773ed87c6d670f3a5ee1438a114b198aad0b6767969dfdab4b
6
7 C:\>openssl rand -hex 64
8 c9157b55870206b2acfc9149530ef6f5a9edda45e9970e88b68e153446f873062dd56bd9318e9c1e
9 bc5410a873fc7e7fba7ff9d45b4f907d2f8719d1ab7d0f54
10
11 C:\>openssl rand -base64 64
12 ruZqMy+qjAdmQgWP2bhpNoXSwhJsB7WrY5IAiPzMpXsG9oYRMuELBccoLVGg1clyU4BTR4AfzLxqMi+00MHbog==
13
14 C:\>openssl rand -base64 -out random.txt 64
15
16 C:\>type random.txt
17 E1fK2vu0Y5h8xPtJRruUUvUO96XYi9KLYOlpf7mwE6pUW3gygmtq+UZxRpaS701+O86235HIYCaqZ8jk4b/UYQ==
18
19 C:\>openssl rand -base64 -rand random.txt 64
20 r2byK4Z+5GdfH+1S3V2OWRCSLklNX2g4v0YH/ms7FR2zVhbykuZY6RvHRtXjpDSeccle1Udz6tFnyVJh2PkcNg==
```

5.4.4. Encriptação simétrica e os algoritmos disponíveis

O `OpenSSL` oferece uma variedade de funções de encriptação simétrica. A Listagem 5.5 mostra a saída do comando `enc -ciphers`, em que se pode observar grupos de funções bem definidos, com algoritmos criptográficos, tamanhos de chaves e modos de operação. Em particular, observam-se os seguintes algoritmos criptográficos: AES, ARIA, blowfish (bf), camelia, cast, chacha20, DES, IDEA, RC4, entre outros. Todas estas funções de encriptação simétrica estão disponíveis pela CLI do `OpenSSL`. As próximas seções mostram usos de algumas delas.

Listagem 5.5. Lista dos algoritmos simétricos de encriptação do `OpenSSL`.

```
C:\>openssl enc -ciphers
Supported ciphers:
-aes-128-cbc               -aes-128-cfb               -aes-128-cfb1
-aes-128-cfb8              -aes-128-ctr               -aes-128-ecb
-aes-128-ofb               -aes-192-cbc               -aes-192-cfb
-aes-192-cfb1              -aes-192-cfb8              -aes-192-ctr
-aes-192-ecb               -aes-192-ofb               -aes-256-cbc
-aes-256-cfb               -aes-256-cfb1              -aes-256-cfb8
-aes-256-ctr               -aes-256-ecb               -aes-256-ofb
-aes128                    -aes128-wrap               -aes192
-aes192-wrap               -aes256                    -aes256-wrap
-aria-128-cbc              -aria-128-cfb              -aria-128-cfb1
-aria-128-cfb8             -aria-128-ctr              -aria-128-ecb
-aria-128-ofb              -aria-192-cbc              -aria-192-cfb
-aria-192-cfb1             -aria-192-cfb8             -aria-192-ctr
-aria-192-ecb              -aria-192-ofb              -aria-256-cbc
-aria-256-cfb              -aria-256-cfb1             -aria-256-cfb8
-aria-256-ctr              -aria-256-ecb              -aria-256-ofb
-aria128                   -aria192                   -aria256
-bf                        -bf-cbc                    -bf-cfb
-bf-ecb                    -bf-ofb                    -blowfish
-camellia-128-cbc          -camellia-128-cfb          -camellia-128-cfb1
-camellia-128-cfb8         -camellia-128-ctr          -camellia-128-ecb
-camellia-128-ofb          -camellia-192-cbc          -camellia-192-cfb
-camellia-192-cfb1         -camellia-192-cfb8         -camellia-192-ctr
-camellia-192-ecb          -camellia-192-ofb          -camellia-256-cbc
-camellia-256-cfb          -camellia-256-cfb1         -camellia-256-cfb8
-camellia-256-ctr          -camellia-256-ecb          -camellia-256-ofb
-camellia128               -camellia192               -camellia256
-cast                      -cast-cbc                  -cast5-cbc
-cast5-cfb                 -cast5-ecb                 -cast5-ofb
-chacha20                  -des                       -des-cbc
-des-cfb                   -des-cfb1                  -des-cfb8
-des-ecb                   -des-ede                   -des-ede-cbc
-des-ede-cfb               -des-ede-ecb               -des-ede-ofb
-des-ede3                  -des-ede3-cbc              -des-ede3-cfb
-des-ede3-cfb1             -des-ede3-cfb8             -des-ede3-ecb
-des-ede3-ofb              -des-ofb                   -des3
-des3-wrap                 -desx                      -desx-cbc
-id-aes128-wrap            -id-aes128-wrap-pad        -id-aes192-wrap
-id-aes192-wrap-pad        -id-aes256-wrap            -id-aes256-wrap-pad
-id-smime-alg-CMS3DESwrap  -idea                      -idea-cbc
-idea-cfb                  -idea-ecb                  -idea-ofb
-rc2                       -rc2-128                   -rc2-40
-rc2-40-cbc                -rc2-64                    -rc2-64-cbc
-rc2-cbc                   -rc2-cfb                   -rc2-ecb
-rc2-ofb                   -rc4                       -rc4-40
-seed                      -seed-cbc                  -seed-cfb
-seed-ecb                  -seed-ofb                  -sm4
-sm4-cbc                   -sm4-cfb                   -sm4-ctr
-sm4-ecb                   -sm4-ofb
```

5.4.5. Encriptação e decriptação com AES

A Listagem 5.6 mostra várias maneiras de encriptar e decriptar o arquivo de exemplo `gabarito.txt` com o comando `enc` e o algoritmo AES. O comando da linha 1 encripta (opção `-c`), codifica o criptograma de saída em Base64 (opção `-a`) e mostra os parâmetros (opção `-p`) da encriptação com AES em modo CTR e chave de 256 bits (`-aes-256-ctr`). Ainda neste comando, a chave é gerada a partir de uma senha (opção `-k`) com o método PBKDF2 (opção `-pbkdf2`). Os parâmetros mostrados em hexadecimal são o `salt` usado no PBKDF2, a chave criptográfica (`key`) e o vetor de inicialização (`iv`) usado como contador do modo CTR. O criptograma de saída é gravado no arquivo `gabarito.aes`. Se a senha não é passada com a opção (`-k`), o `OpenSSL` pede que uma senha seja digitada.

O comando `enc -d`, das linhas 6 e 7, faz a decriptação do criptograma contido no arquivo `gabarito.aes`, nas mesmas configurações utilizadas pela encriptação, gerando o arquivo de saída `gabarito4.txt`, cujo conteúdo é mostrado pelo comando `type` na linha 9 e consiste do texto claro recuperado.

O comando `enc -c`, das linhas de 14 até 16, não utiliza mais as opções `-pbkdf2` e `-k` (em minúsculo) para geração de chaves a partir de senhas. Em vez disto, a chave criptográfica é passada explicitamente no comando com a opção `-K` (em maiúsculo) seguida pela chave em hexadecimal (linha 15). O mesmo acontece com o IV passado com a opção `-iv` (linha 16). A opção `-salt` não é mais necessária uma vez que não há mais geração de parâmetros aleatórios internos a função. Vale lembrar que esta chave criptográfica pode ter sido gerada, por exemplo, pelo comando `rand` explicado anteriormente.

O comando `enc -d`, das linhas de 18 até 20, faz a decriptação do arquivo `gabarito.aes` nas mesmas configurações utilizadas na encriptação anterior, gerando o arquivo de saída `gabarito5.txt`, cujo conteúdo é mostrado pelo comando na linha 22.

Listagem 5.6. Encriptação com AES.

```
1 C:\>openssl enc -e -a -p -aes-256-ctr -pbkdf2 -in gabarito.txt -out gabarito.aes -k 1234
2 salt=BEF1B00E6BCC6C01
3 key=6CB40CAEF6FDEDB31DBD908256F63276DB8FBC1E3430CEF18B6E0F5CD112D5F2
4 iv =E570548E2B1376147E3342F08082E29A
5
6 C:\>openssl enc -d -a -salt -aes-256-ctr -pbkdf2 -in gabarito.aes -out gabarito4.txt
7  -k 1234
8
9 C:\>type gabarito4.txt
10 Gabarito da prova
11 1-a 2-c 3-d 4-c 5-a 6-b 7-d 8-d 9-a 10-c
12
13 C:\>openssl enc -e -a -aes-256-ctr -in gabarito.txt -out gabarito.aes
14  -K 6CB40CAEF6FDEDB31DBD908256F63276DB8FBC1E3430CEF18B6E0F5CD112D5F2
15  -iv E570548E2B1376147E3342F08082E29A
16
17 C:\>openssl enc -d -a -aes-256-ctr -in gabarito.aes -out gabarito5.txt
18  -K 6CB40CAEF6FDEDB31DBD908256F63276DB8FBC1E3430CEF18B6E0F5CD112D5F2
19  -iv E570548E2B1376147E3342F08082E29A
20
21 C:\>type gabarito5.txt
22 Gabarito da prova
23 1-a 2-c 3-d 4-c 5-a 6-b 7-d 8-d 9-a 10-c
```

5.4.6. Preenchimento de blocos na criptografia simétrica

Na encriptação de blocos, o texto claro deve ser um múltiplo do tamanho do bloco do algoritmo criptográfico. Quando o comprimento do texto claro não é múltiplo do tamanho do bloco, um método de (*padding*) é usado para preencher o último bloco incompleto do texto claro. A Listagem 5.7 mostra a encriptação sem *padding* e a opção `-nopad`. As linhas de 1 até 7 listam dois arquivos com bloco incompleto, um com 1 byte e outro com 15 bytes, mais um arquivo de bloco completo (16 bytes). O comando da linha 8 encripta com AES-128-ECB e `-nopad` um texto claro múltiplo do bloco. A decriptação está na linha 9. Todos os comandos usam a mesma chave.

Os comandos das linhas 12 e 13 encriptam com *padding* e decriptam sem *padding* um arquivo de 1 byte, enquanto os comandos das linhas 15 e 16 fazem o mesmo para um arquivo de 15 bytes. O resultado nos dois casos é um arquivo decriptado maior que o original e com tamanho de um bloco. Os bytes extras são o *padding*, mostrados na Figura 5.14, tela do software HxD (`https://mh-nexus.de/en/hxd`). Já os comandos das linhas 18 e 19 encriptam com *padding* e decriptam sem *padding* um arquivo de 16 bytes (um bloco) e o resultado da decriptação é um arquivo com dois blocos, o bloco extra é o *padding* (vide Figura 5.14).

Listagem 5.7. Prenchimento de blocos do AES.

```
1 C:\>type i1.txt
2 1
3 C:\>type iF.txt
4 FFFFFFFFFFFFFFF
5 C:\>type i16F.txt
6 0123456789ABCDEF
7
8 C:\>openssl enc -
      e -AES-128-ECB -in i16F.txt -K 6CB40CAEF6FDEDB31DBD908256F6327A -nopad -out o16F.txt
9 C:\>openssl enc -d -AES-128-ECB -in o16F.txt -K 6CB4...327A -nopad
10 0123456789ABCDEF
11
12 C:\>openssl enc -e -AES-128-ECB -in i1.txt -K 6CB4...327A -out o1.txt
13 C:\>openssl enc -d -AES-128-ECB -in o1.txt -K 6CB4...327A -out o1_2.txt -nopad
14
15 C:\>openssl enc -e -AES-128-ECB -in iF.txt -K 6CB4...327A -out oF.txt
16 C:\>openssl enc -d -AES-128-ECB -in oF.txt -K 6CB4...327A -out oF_2.txt -nopad
17
18 C:\>openssl enc -e -AES-128-ECB -in i16F.txt -K 6CB4...327A -out o16F.txt
19 C:\>openssl enc -d -AES-128-ECB -in o1F.txt -K 6CB4...327A -out o16F_2.txt -nopad
```

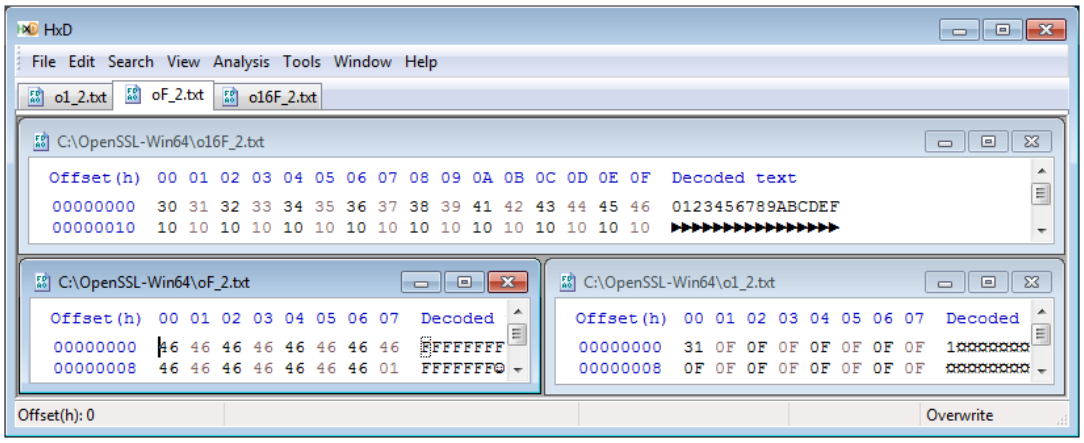


Figura 5.14. Preenchimentos dos blocos parciais e do bloco completo.

5.4.7. Repetição de padrões na encriptação de blocos

Na encriptação simétrica de blocos, padrões presentes no texto claro podem gerar padrões identificáveis no criptograma. Este comportamento pode ser exemplificado pela encriptação simétrica no modo de operação ECB, no comando a seguir, onde a arte em ASCII de um disco flexível (adaptada de `https://www.asciiart.eu`) é encriptada com AES-128-ECB. A Figura 5.15 mostra, no criptograma, os padrões equivalentes aos encontrados no texto claro.

```
C:\>openssl enc -e -AES-128-ECB -in floppy.txt -out floppypattern.txt

C:\>type floppy.txt
..............................................................................................
..............................................................................................
................aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa..............
................8888888888888888888888888888888888888888888888888888888888888888..............
................8888""""""""""""888888888888888888888888888888888888888888888888..............
................8888            888888888888888888888888888888888888888888888888..............
................8888            888888888888888888888888888888888888888888888888..............
................8888            88888888888888888888888888888888888888888888888"..............
................8888aaaaaaaaaaaa88888888888888888888888888888888888888888888888a..............
................8888888888888888888888888888888888888888888888888888888888888888..............
................8888888888888888888888888888888888888888888888888888888888888888..............
................8888888888888888888888888888888888888888888888888888888888888888..............
................88888888888888888888888":::::"8888888888888888888888888888888888..............
................888888888888888888888::;gPPRg;::88888888888888888888888888888888..............
................888888888888888888888::dP’  `Yb::8888888888888888888888888888888..............
................888888888888888888888::8)    (8::8888888888888888888888888888888..............
................88888888888888888888;:Yb     dP:;88( )88888888888888888888888888..............
................888888888888888888888;:"8ggg8":;88888888888888888888888888888888..............
................88888888888888888888888aa:::aa8888888888888888888888888888888888..............
................8888888888888888888888888888888888888888888888888888888888888888..............
................8888888888888888888888888888888888888888888888888888888888888888..............
................88888888888888888888888888"8888888888888888888888888888888888888..............
................8888888888888888888888888:::888888888888888888888888888888888888..............
................8888888888888888888888888:::888888888888888888888888888888888888..............
................8888888888888888888888888:::888888888888888888888888888888888888..............
................8888888888888888888888888:::888888888888888888888888888888888888..............
................8888888888888888888888888:::888888888888888888888888888888888888..............
................88888888888888888888888888a8888888888888888888888888888888888888..............
................"""""""""""""""""""'  `""""""""'  `"""""""""""""""""""""""""""""..............
..............................................................................................
..............................................................................................
```

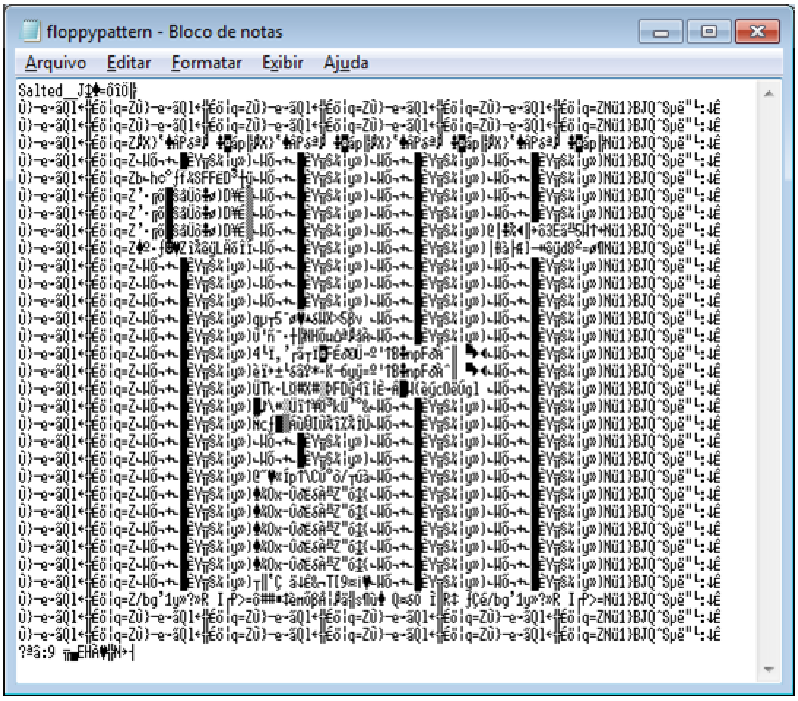


Figura 5.15. Padrões do texto claro geram padrões no criptograma com modo ECB.

5.4.8. Os modos de operação da encriptação simétrica de blocos

A Listagem 5.8 ilustra as diferenças de tolerância a falhas entre os modos de operação da encriptação simétrica, em particular, o algoritmo AES e os modos de operação ECB, CBC, CFB, OFB e CTR. O comando `type` (na linha 1) mostra o conteúdo do arquivo usado no teste. Todos os comandos de encriptação `enc -e` (linhas de 7 até 17) usam a mesma chave criptográfica, o mesmo arquivo de entrada e, quando necessário, o mesmo vetor de inicialização. A diferença entre eles está no modo de operação do AES: o comando da linha 4 usa o modo ECB (`-aes-128-ecb`), o comando da linha 7 usa o modo CBC (`-aes-128-cbc`), o comando da linha 10 usa o CFB (`-aes-128-cfb`), o comando da linha 13 usa o OFB (`-aes-128-ofb`) e a linha 10 usa o CTR (`-aes-128-ctr`).

Na linha 19, os arquivos de saída dos comandos `enc -e` são editados (`edit`), simulando a adulteração do criptograma: trocar o primeiro caractere pela próxima letra (p.ex., 'y' é substituído por 'z' e 'L' é substituído por 'M'), causando uma mudança no criptograma. Os comandos de decriptação `end -d` reagem de formas diferentes a adulteração: o comando no modo ECB (linha 21) perde todo o primeiro bloco do criptograma; o comando no modo CBC (linha 24) perde o primeiro bloco e o primeiro caractere do segundo bloco; o modo CFB (linha 28) perde o primeiro caractere do primeiro bloco e o segundo bloco inteiro; o modo OFB (linha 32) e o modo CTR (linha 36) perdem só o primeiro caractere do primeiro bloco.

Listagem 5.8. Os modos de operação e como toleram erros no criptograma.

```
1 C:\>type alex.txt
2 ALEXANDRE MELO BRAGA ALEXANDRE MELO BRAGA
3
4 C:\> openssl enc -e -a -aes-128-ecb -in alex.txt -K 6CB40CAEF6FDEDB31DBD908256F63276
5  -out alex.ecb
6
7 C:\> openssl enc -e -a -aes-128-cbc -in alex.txt -K 6CB40CAEF6FDEDB31DBD908256F63276
8  -iv E570548E2B1376147E3342F08082E29A -out alex.cbc
9
10 C:\> openssl enc -e -a -aes-128-cfb -in alex.txt -K 6CB40CAEF6FDEDB31DBD908256F63276
11  -iv E570548E2B1376147E3342F08082E29A -out alex.cfb
12
13 C:\> openssl enc -e -a -aes-128-ofb -in alex.txt -K 6CB40CAEF6FDEDB31DBD908256F63276
14 -iv E570548E2B1376147E3342F08082E29A -out alex.ofb
15
16 C:\> openssl enc -e -a -aes-128-ctr -in alex.txt -K 6CB40CAEF6FDEDB31DBD908256F63276
17  -iv E570548E2B1376147E3342F08082E29A -out alex.ctr
18
19 C:\>edit alex.ecb alex.cbc alex.cfb alex.ofb alex.ctr
20
21 C:\> openssl enc -d -a -aes-128-ecb -in alex.ecb -K 6CB40CAEF6FDEDB31DBD908256F63276
22 ****************RAGA ALEXANDRE MELO BRAGA
23
24 C:\> openssl enc -d -a -aes-128-cbc -in alex.cbc -K 6CB40CAEF6FDEDB31DBD908256F63276
25  -iv E570548E2B1376147E3342F08082E29A
26 ******************AGA ALEXANDRE MELO BRAGA
27
28 C:\> openssl enc -d -a -aes-128-cfb -in alex.cfb -K 6CB40CAEF6FDEDB31DBD908256F63276
29  -iv E570548E2B1376147E3342F08082E29A
30 ELEXANDRE MELO B****************ELO BRAGA
31
32 C:\> openssl enc -d -a -aes-128-ofb -in alex.ofb -K 6CB40CAEF6FDEDB31DBD908256F63276
33 -iv E570548E2B1376147E3342F08082E29A
34 *LEXANDRE MELO BRAGA ALEXANDRE MELO BRAGA
35
36 C:\> openssl enc -d -a -aes-128-ctr -in alex.ctr -K 6CB40CAEF6FDEDB31DBD908256F63276
37  -iv E570548E2B1376147E3342F08082E29A
38 *LEXANDRE MELO BRAGA ALEXANDRE MELO BRAGA
```

5.4.9. Funções do `OpenSSL` para cálculo de *hash* e MAC

A Listagem 5.9 mostra algumas maneiras de calcular valores *hash* e *tags* de autenticação MAC com o comando `dgst` do `OpenSSL`. Os primeiros quatro comandos calculam *hashes*. O comando `dgst` na linha 1 calcula o valor *hash* do arquivo de exemplo `gabarito.txt` com a função de resumo criptográfico SHA-256 (opção `-sha256`) e imprime no terminal o valor em hexadecimal (opção `-r`).

O comando `dgst` na linha 4 calcula o mesmo valor de *hash* do arquivo `gabarito.txt` com opção `-sha256`, mas não usa a opção `-r` e mostra o mesmo hexadecimal em uma linha formatada de modo diferente. Este exemplo ilustra o determinismo das funções de *hash* criptográfico. Isto é, mesmo texto claro de entrada e mesma função de *hash* resultam no mesmo valor *hash*, apesar das diferenças de apresentação.

O comando `dgst` na linha 7 calcula o valor de *hash* do mesmo arquivo `gabarito.txt` com a função de resumo criptográfico SHA-512 (opção `-sha512`), resultando em um valor diferente do anterior e com o dobro de tamanho.

O comando `dgst` na linha 11 calcula o valor de *hash* do mesmo arquivo `gabarito.txt` com outra função de resumo criptográfico de 512 bits, o SHA3 (opção `-sha3-512`), resultando em uma valor diferente do anterior, mas com o mesmo tamanho.

Os dois últimos comandos da Listagem 5.9 calculam *tags* de autenticação do tipo HMAC. O comando `dgst` da linha 15 calcula a *tag* de 256 bits para o arquivo `gabarito.txt` com as opções `-hmac` e `-sha256`. Vale observar que a opção `-hmac` é acompanhada pelo valor hexadecimal da chave criptográfica utilizada pela função HMAC. Esta chave pode ter sido gerada, por exemplo, pelo comando `rand` mostrado anteriormente.

Finalmente, o comando `dgst` da linha 19 calcula uma *tag* de 512 bits para o arquivo `gabarito.txt` com as opções `-sha512` e `-hmac`, utilizando a mesma chave criptográfica do exemplo anterior, e resulta em uma *tag* diferente da anterior, uma vez que a função de resumo criptográfico é diferente.

Listagem 5.9. Cálculo de *Hashes* e MACs.

```
1 C:\>openssl dgst -sha256 -r gabarito.txt
2 385fb55c7aed9ec76dc31b8392a90a61ba5acd1e578112a1977047f9d1e9c3b7 *gabarito.txt
3
4 C:\>openssl dgst -sha256 gabarito.txt
5 SHA256(gabarito.txt)= 385fb55c7aed9ec76dc31b8392a90a61ba5acd1e578112a1977047f9d1e9c3b7
6
7 C:\>openssl dgst -sha512 -r gabarito.txt
8 0cd7521e631ec426389e5ced1c9079486e5f733a6ab2545ff360333a259c9d3d9bb8659cfa32619df0e9dc
9 778d8a57988a681be5dc0d7942938fdefc9367b4ea *gabarito.txt
10
11 C:\>openssl dgst -sha3-512 -r gabarito.txt
12 2e3d039e704686479e7ad3145376232O4b68e1cb89739baa41a43650ac12f32b0bb76533a8045c5dde8a93
13 acb6743f8443b2c4dc43cdab10e78ab596167337a3 *gabarito.txt
14
15 C:\>openssl dgst -sha256 -hmac 3f44a88d098cdb8a384922e88a30dbe67f7178fd gabarito.txt
16 HMAC-SHA256(gabarito.txt)= f59848834988db68167a7a5527273ed494dbd6374cafefd0c5c3f7f18
17 dd7a02c
18
19 C:\>openssl dgst -sha3-512 -hmac 3f44a88d098cdb8a384922e88a30dbe67f7178fd gabarito.txt
20 HMAC-SHA3-512(gabarito.txt)= baf213dcc21d1bf9337be63495ceff4dd553c7cbb47da89818f7b9cf
21 ccdf336fe25c28c56e9f07062e74c83b57542b0df96223ed2af0765c9a4b678200160d80
```

5.4.10. Encriptação e decriptação assimétricas com RSA

Esta seção mostra a geração de um par de chaves assimétricas para o algoritmo RSA e dois exemplos de encriptação utilizado o mesmo par de chaves. Na Listagem 5.10, o comando `genrsa` na linha 1 gera chaves RSA de 2048 bits e usa a opção `-rand` para alimentar o gerador de números pseudo-aleatórios com uma semente calculada anteriormente. A opção `-rand`, quando bem utilizada, ajuda a evitar a geração de chaves RSA com parâmetros de baixa entropia, uma fonte comum de vulnerabilidades de implantação. Além disso, o par de chaves é gravado em um arquivo de saída (opção `-out chaves_rsa`) encriptado com AES (opção `-aes-256-ctr`) e chave derivada a partir de uma senha.

O comando `rsa` da linha 9 é usado para exportar a chave pública (opção `-pubout`) contida no par de chaves gerado anteriormente (opção `-in chaves_rsa`) em um arquivo separado (opção `-out chave_pub.rsa`). Já o comando `rsa` da linha 13 é usado para exportar a chave privada contida no par de chaves gerado anteriormente (opção `-in chaves_rsa`) em um arquivo separado (opção `-out chave_priv.rsa`). Vale observar a usabilidade ruim do `OpenSSL`: se a opção `-pubout` não é usada, o comportamento padrão (inseguro) é exportar a chave privada.

O comando `rsautl` da linha 17 encripta com RSA-OAEP (opções `-encrypt -oaep`) e a chave pública (opção `-inkey chave_pub.rsa`). O comando `rsautl` da linha 20 decripta (`-decrypt -oaep`) com a chave privada (`-inkey chave_priv.rsa`).

Na linha 23, o comando `req` extrai a chave pública de um certificado auto-assinado. Esta chave é usada no comando `smime` da linha 20 para encriptar uma chave simétrica temporária (opção `-aes-128-cbc`) que encripta o texto claro. Já a linha 27 decripta com a chave privada a chave temporária embutida no criptograma que é usada para decriptar o texto claro.

Listagem 5.10. Encriptação com RSA-OAEP e chave de transporte SMIME.

```
1 C:\>openssl genrsa -rand random.txt -aes-256-ctr -out chaves_rsa 2048
2 Generating RSA private key, 2048 bit long modulus (2 primes)
3 ..................................................+++++
4 .............+++++
5 e is 65537 (0x010001)
6 Enter pass phrase for chaves_rsa:
7 Verifying - Enter pass phrase for chaves_rsa:
8
9 C:\>openssl rsa -in chaves_rsa -pubout -out chave_pub.rsa
10 Enter pass phrase for chaves_rsa:
11 writing RSA key
12
13 C:\>openssl rsa -in chaves_rsa -out chave_priv.rsa
14 Enter pass phrase for chaves_rsa:
15 writing RSA key
16
17 C:\>openssl rsautl -encrypt -oaep -in gabarito.txt -pubin -inkey chave_pub.rsa
18 -out gabarito.rsac
19
20 C:\>openssl rsautl -decrypt -oaep -in gabarito.rsac -inkey chave_priv.rsa
21  -out gabarito6.txt
22
23 C:\>openssl req -new -key chaves_rsa -x509 -out raiz.crt
24
25 C:\>openssl smime -in gabarito.txt -encrypt -out gabarito.smime -aes-128-cbc raiz.crt
26
27 C:\>openssl smime -decrypt -in gabarito.smime -inkey chave_priv.rsa
28 Gabarito da prova
29 1-a 2-c 3-d 4-c 5-a 6-b 7-d 8-d 9-a 10-c
```

5.4.11. Duas versões de RSA PKCS#1: v1.5 e v2.0

A Listagem 5.11 mostra dois exemplos de encriptação assimétrica com RSA. O primeiro sem a opção `-oaep` e o segundo com este opção explícita. A ausência da opção `-oaep` na encriptação RSA com o comando `rsautl` faz com que o preenchimento (*padding*) PKCS#1 v1.5 seja utilizado. Este formato de preenchimento é, em certos casos, considerado inseguro [Bleichenbacher 1998], devido a sua susceptibilidade ao ataque de *padding oracle*, mas que ainda é bastante utilizado no SSL/TLS até a versão v1.2, Mostrando como é difícil seguir as boas práticas e ao mesmo tempo manter a compatibilidade com o passado.

Tanto a PKCS#1 v1.5, quanto o RSA-OAEP (PKCS#1 v2.0) limitam o tamanho do texto claro que pode ser encriptado em uma única chamada do comando `rsautl`. Este limite está relacionado ao tamanho do corpo finito (e da chave) usado na aritmética modular do algoritmo RSA e, no cado o RSA-OAEP, pode ser determinado, em bytes, pela fórmula *(ks-2*hs)/8–2*, onde *ks* é o tamanho da chave RSA em bits e *hs* é o tamanho do *hash* em bits usado pelo *padding* OAEP.

Na Listagem 5.11, os comandos das linhas 5 e 6 terminam com sucesso a encriptação do arquivo curto, enquanto os comandos de encriptação do arquivo médio falham em parte, por que o comando da linha 16 (com `-oaep`) já não consegue processar tantos dados de entrada. Já ambos os comandos de encriptação do arquivo longo falham devido ao excesso de dados de entrada (apenas 50 caracteres a mais que no caso anterior), conforme as mensagens de erro emitidas.

Listagem 5.11. Limite de tamanho para o criptograma no RSA.

```
1 C:\>type curto.txt
2 Gabarito da prova:
3 01-a 02-c 03-d 04-c 05-a 06-b 07-d 08-d 09-a 10-c
4
5 C:\>openssl rsautl -encrypt -in curto.txt -pubin -inkey chave_pub.rsa -out o.rsac
6 C:\>openssl rsautl -encrypt -oaep -in curto.txt -pubin -inkey chave_pub.rsa -out o.oaep
7
8 C:\>type medio.txt
9 Gabarito da prova:
10 01-a 02-c 03-d 04-c 05-a 06-b 07-d 08-d 09-a 10-c
11 11-a 12-c 13-d 14-c 15-a 16-b 17-d 18-d 19-a 20-c
12 21-a 22-c 23-d 24-c 25-a 26-b 27-d 28-d 29-a 30-c
13 31-a 32-c 33-d 34-c 35-a 36-b 37-d 38-d 39-a 40-c
14
15 C:\>openssl rsautl -encrypt -in medio.txt -pubin -inkey chave_pub.rsa -out o.rsac
16 C:\>openssl rsautl -encrypt -oaep -in medio.txt -pubin -inkey chave_pub.rsa -out o.oaep
17 RSA operation error
18 2636:error:0409A06E:rsa routines:RSA_padding_add_PKCS1_OAEP_mgf1
19 :data too large for key size:crypto\rsa\rsa_oaep.c:62:
20
21 C:\>type longo.txt
22 Gabarito da prova:
23 01-a 02-c 03-d 04-c 05-a 06-b 07-d 08-d 09-a 10-c
24 11-a 12-c 13-d 14-c 15-a 16-b 17-d 18-d 19-a 20-c
25 21-a 22-c 23-d 24-c 25-a 26-b 27-d 28-d 29-a 30-c
26 31-a 32-c 33-d 34-c 35-a 36-b 37-d 38-d 39-a 40-c
27 41-a 42-c 43-d 44-c 45-a 46-b 47-d 48-d 49-a 50-c
28 C:\>openssl rsautl -encrypt -in longo.txt -pubin -inkey chave_pub.rsa -out o.rsac
29 RSA operation error
30 6504:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2
31 :data too large for key size:crypto\rsa\rsa_pk1.c:125:
32
33 C:\>openssl rsautl -encrypt -oaep -in longo.txt -pubin -inkey chave_pub.rsa -out o.rsac
34 RSA operation error
35 8988:error:0409A06E:rsa routines:RSA_padding_add_PKCS1_OAEP_mgf1
36 :data too large for key size:crypto\rsa\rsa_oaep.c:62:
```

5.4.12. Geração de certificados digitais

A Listagem 5.12 contém os comandos do `OpenSSL` para gerar um par de chaves RSA, gerar um certificado digital auto-assinado para a chave pública do par (ilustrando uma Autoridade Certificadora - CA), gerar um par de chaves para um usuário final e assinar o certificado digital deste usuário com a chave privada da CA.

O comando `genrsa` da linha 1 gera as chaves de 2048 bits da CA e as armazena em um arquivo encriptado com `-aes-128-cbc` e uma chave derivada de senha. Já o comando `req -new` da linha 9 gera uma requisição para um certificado X.509 auto-assinado da chave pública da CA. O outro comando `genrsa` da linha 13 gera um par de chaves RSA de 2048 bits para um usuário final, enquanto o comando `req -new` da linha 21 gera uma requisição de certificado para a chave pública do usuário. Uma vez que a validade não foi informada, estes certificados tem validade *default* de um mês (30 dias).

Finalmente, o comando `x509` da linha 25 recebe a requisição de assinatura de certificado (opção `-in alex.csr`), e a assina com a chave privada da CA (opção `-CAkey CA_chaves_rsa`), resultado no certificado digital do usuário final (opção `-out alex.crt`) com número de série 3. Os comandos `verify` das linhas 32 e 35 verificam as assinaturas digitais, respectivamente, dos certificados da CA e do usuário final.

Listagem 5.12. Certificados digitais para chaves RSA.

```
1 C:\>openssl genrsa -aes-128-cbc -out CA_chaves_rsa
2 Generating RSA private key, 2048 bit long modulus (2 primes)
3 ......+++++
4 ..............................................................+++++
5 e is 65537 (0x010001)
6 Enter pass phrase for CA_chaves_rsa:
7 Verifying - Enter pass phrase for CA_chaves_rsa:
8
9 C:\>openssl req -new -key CA_chaves_rsa -x509 -out CAraiz.crt
10 Enter pass phrase for CA_chaves_rsa:
11 #[]...
12
13 C:\>openssl genrsa -aes-128-cbc -out alex_rsa
14 Generating RSA private key, 2048 bit long modulus (2 primes)
15 ......+++++
16 ..............+++++
17 e is 65537 (0x010001)
18 Enter pass phrase for alex_rsa:
19 Verifying - Enter pass phrase for alex_rsa:
20
21 C:\>openssl req -new -key alex_rsa -out alex.csr
22 Enter pass phrase for alex_rsa:
23 #...
24
25 C:\>openssl x509 -req -in alex.csr -CA CAraiz.crt -CAkey CA_cha
26 ves_rsa -out alex.crt -set_serial 3
27 Signature ok
28 subject=C = br, ST = sp, L = cps, O = alex, CN = www.alex.com.br
29 Getting CA Private Key
30 Enter pass phrase for CA_chaves_rsa:
31
32 C:\>openssl verify -CAfile CAraiz.crt CAraiz.crt
33 CAraiz.crt: OK
34
35 C:\>openssl verify -CAfile CAraiz.crt alex.crt
36 alex.crt: OK
```

5.4.13. Assinaturas digitais com RSA

A Listagem 5.13 mostra os comandos do `OpenSSL` para geração de assinaturas digitais e verificação destas assinaturas. Neste exemplo, a assinatura é computada com o algoritmo RSA e chaves de 2048 bits sobre o *hash* do arquivo de entrada.

O comando `dgst` na linha 1 gera uma assinatura digital do arquivo `gabarito.txt` com a chave privada (opção `-sign alex_rsa`). A assinatura é gerada sobre um *hash* de 512 bits do arquivo de entrada (opção `-sha512`). O resultado é mostrado no terminal em hexadecimal (opção `-hex`), onde se vê que a assinatura tem 2048 bits de tamanho.

O comando `dgst` da linha 11 repete o processo de assinatura do comando anterior, mostrando que este método de assinatura digital com RSA é determinístico. Isto é, se forem fornecidos o mesmo arquivo de entrada, a mesma chave privada e a mesma função de *hash*, então a mesma assinatura será gerada.

Já o comando `dgst` da linha 21 não usa a opção `-hex` e grava a assinatura em arquivo de saída. O comando `rsa` da linha 24 exporta a chave pública em um arquivo separado para que ela seja usada na verificação de assinaturas geradas pela chave privada correspondente. Finalmente, na linha 26, o comando `dgst` usa a chave pública para verificar (opção `-verify alex_pub`) a assinatura (opção `-signature gabarito.sign`) do arquivo de entrada `gabarito.txt`. A verificação é bem-sucedida.

Listagem 5.13. Assinaturas digitais e verificação de assinaturas com RSA.

```
1 C:\>openssl dgst -sha512 -hex -sign alex_rsa gabarito.txt
2 Enter pass phrase for alex_rsa:
3 RSA-SHA512(gabarito.txt)= 0b84698f3e0ef7279638819731532e2f65f84e7013cab1193f0582
4 84f022433d90cc5978ea3cc8486662fb48884d072a0ec1e0cf4ddb75f8503ff3f564125303d4d861
5 b8bd718e5eab85b5902f1881845d11c417d1c5796959e5d4baee4da3216e0ba210d6ce583ca7ea53
6 83f9ab727991a7e6404babc7cfe2fc28885e126a36db28a502f2ae76dde734086686806550c5e20c
7 e2fe1bea67a09b6caf62c176df40a56d54470f86b739539d3e5409dc00a92324cedd4737228b653c
8 18ee84d2b81b49eabdb97ff2e7d802a44eaa3de12c6b684e4be88a81008e9bb776c4e3c1e9144e0c
9 c0acbc702ff35904d2b30c0e2e0a1e26344ce6cb2fd96c1e8f3ecbba11
10
11 C:\>openssl dgst -sha512 -hex -sign alex_rsa gabarito.txt
12 Enter pass phrase for alex_rsa:
13 RSA-SHA512(gabarito.txt)= 0b84698f3e0ef7279638819731532e2f65f84e7013cab1193f0582
14 84f022433d90cc5978ea3cc8486662fb48884d072a0ec1e0cf4ddb75f8503ff3f564125303d4d861
15 b8bd718e5eab85b5902f1881845d11c417d1c5796959e5d4baee4da3216e0ba210d6ce583ca7ea53
16 83f9ab727991a7e6404babc7cfe2fc28885e126a36db28a502f2ae76dde734086686806550c5e20c
17 e2fe1bea67a09b6caf62c176df40a56d54470f86b739539d3e5409dc00a92324cedd4737228b653c
18 18ee84d2b81b49eabdb97ff2e7d802a44eaa3de12c6b684e4be88a81008e9bb776c4e3c1e9144e0c
19 c0acbc702ff35904d2b30c0e2e0a1e26344ce6cb2fd96c1e8f3ecbba11
20
21 C:\>openssl dgst -sha512 -sign alex_rsa -out gabarito.sign  gabarito.txt
22 Enter pass phrase for alex_rsa:
23
24 C:\>openssl rsa -in alex_rsa -pubout -out alex_pub
25
26 C:\>openssl dgst -sha512 -verify alex_pub -signature gabarito.sign gabarito.txt
27 Verified OK
```

5.4.14. Curvas elípticas no `OpenSSL`

O `OpenSSL` oferece diversas opções de curvas elípticas. A Listagem 5.14 contém as curvas elípticas sobre corpos primos F_p e binários F_{2^m} disponíveis no `OpenSSL`, e inclui aquelas padronizadas e ainda consideradas seguras pelos padrões internacionais [NIST 2013, SEC 2010]. Algumas curvas foram omitidas por falta de espaço. A nomenclatura das curvas segue o padrão SEC-2 [SEC 2010], com nomes no formato `sec[p|t]XXX[r|k]v`, onde `[p|t]` indica uma curva sobre corpo primo (`p`) ou binário (`t`), enquanto o número `XXX` indica o tamanho em bits da chave e a letra `[r|k]` indica que a curva usa os parâmetros de Koblitz (`k`) ou randômicos (`r`), em uma determinada versão (`v`).

A versão do `OpenSSL` usada neste texto não contém a curva 25519 [Bernstein 2006] disponível na CLI. Além disto, na Listagem 5.14, as linhas marcadas com sinal de mais (+) indicam as curvas presentes no padrão SEC-2 [SEC 2010]. Já as linhas marcadas com sinal de menos (-) indicam as curvas que são consideradas fracas para os padrões de segurança atuais, com menos de 80 bits de segurança. Por exemplo, estas curvas fizeram parte da versão 1 do padrão SEC-2 [SEC 2000], mas que foram excluídas da versão 2 deste mesmo padrão [SEC 2010]. As curvas marcadas com um asterisco (*) são consideradas inseguras por Daniel Bernstein [Bernstein et al. 2013], que considera estas curvas complexas de implementar, facilitando a ocorrência de defeitos que levam a vulnerabilidades, entre outros problemas.

As curvas fracas ou inseguras não devem mais ser usadas. As curvas com nível de segurança menor ou igual a 80 bits [SEC 2000] são as seguintes: secp112r1, secp112r2, secp128r1, secp128r2, secp160k1, secp160r1, secp160r2, sect113r1, sect113r2 ,sect131r1 e sect131r2. As curvas inseguras [Bernstein et al. 2013] são as seguintes: secp224r1, secp256k1 ,secp384r1, brainpoolP256t1 e brainpoolP384t1. Excluindo-se estas exceções, em princípio, as outras curvas elípticas na Listagem 5.14 podem ser utilizadas.

Listagem 5.14. As curvas elípticas disponíveis no `OpenSSL`.

```
1 C:\>openssl ecparam -list_curves
2  -  secp112r1 : SECG/WTLS curve over a 112 bit prime field
3  -  secp112r2 : SECG curve over a 112 bit prime field
4  -  secp128r1 : SECG curve over a 128 bit prime field
5  -  secp128r2 : SECG curve over a 128 bit prime field
6  -  secp160k1 : SECG curve over a 160 bit prime field
7  -  secp160r1 : SECG curve over a 160 bit prime field
8  -  secp160r2 : SECG/WTLS curve over a 160 bit prime field
9  +  secp192k1 : SECG curve over a 192 bit prime field
10 +  secp224k1 : SECG curve over a 224 bit prime field
11 *+ secp224r1 : NIST/SECG curve over a 224 bit prime field
12 *+ secp256k1 : SECG curve over a 256 bit prime field
13 *+ secp384r1 : NIST/SECG curve over a 384 bit prime field
14 +  secp521r1 : NIST/SECG curve over a 521 bit prime field
15    prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field
16    prime192v2: X9.62 curve over a 192 bit prime field
17    prime192v3: X9.62 curve over a 192 bit prime field
18    prime239v1: X9.62 curve over a 239 bit prime field
19    prime239v2: X9.62 curve over a 239 bit prime field
20    prime239v3: X9.62 curve over a 239 bit prime field
21    prime256v1: X9.62/SECG curve over a 256 bit prime field
22 -  sect113r1 : SECG curve over a 113 bit binary field
23 -  sect113r2 : SECG curve over a 113 bit binary field
24 -  sect131r1 : SECG/WTLS curve over a 131 bit binary field
25 -  sect131r2 : SECG curve over a 131 bit binary field
26 +  sect163k1 : NIST/SECG/WTLS curve over a 163 bit binary field
27 +  sect163r1 : SECG curve over a 163 bit binary field
28 +  sect163r2 : NIST/SECG curve over a 163 bit binary field
29    sect193r1 : SECG curve over a 193 bit binary field
```

```
30      sect193r2 : SECG curve over a 193 bit binary field
31  +   sect233k1 : NIST/SECG/WTLS curve over a 233 bit binary field
32  +   sect233r1 : NIST/SECG/WTLS curve over a 233 bit binary field
33  +   sect239k1 : SECG curve over a 239 bit binary field
34  +   sect283k1 : NIST/SECG curve over a 283 bit binary field
35  +   sect283r1 : NIST/SECG curve over a 283 bit binary field
36  +   sect409k1 : NIST/SECG curve over a 409 bit binary field
37  +   sect409r1 : NIST/SECG curve over a 409 bit binary field
38  +   sect571k1 : NIST/SECG curve over a 571 bit binary field
39  +   sect571r1 : NIST/SECG curve over a 571 bit binary field
40      c2pnb163v1: X9.62 curve over a 163 bit binary field
41      c2pnb163v2: X9.62 curve over a 163 bit binary field
42      c2pnb163v3: X9.62 curve over a 163 bit binary field
43      c2pnb176v1: X9.62 curve over a 176 bit binary field
44      c2tnb191v1: X9.62 curve over a 191 bit binary field
45      c2tnb191v2: X9.62 curve over a 191 bit binary field
46      c2tnb191v3: X9.62 curve over a 191 bit binary field
47      c2pnb208w1: X9.62 curve over a 208 bit binary field
48      c2tnb239v1: X9.62 curve over a 239 bit binary field
49      c2tnb239v2: X9.62 curve over a 239 bit binary field
50      c2tnb239v3: X9.62 curve over a 239 bit binary field
51      c2pnb272w1: X9.62 curve over a 272 bit binary field
52      c2pnb304w1: X9.62 curve over a 304 bit binary field
53      c2tnb359v1: X9.62 curve over a 359 bit binary field
54      c2pnb368w1: X9.62 curve over a 368 bit binary field
55      c2tnb431r1: X9.62 curve over a 431 bit binary field
56      [ ... ]
57      brainpoolP224t1: RFC 5639 curve over a 224 bit prime field
58      brainpoolP256r1: RFC 5639 curve over a 256 bit prime field
59  *   brainpoolP256t1: RFC 5639 curve over a 256 bit prime field
60      brainpoolP320r1: RFC 5639 curve over a 320 bit prime field
61      brainpoolP320t1: RFC 5639 curve over a 320 bit prime field
62      brainpoolP384r1: RFC 5639 curve over a 384 bit prime field
63  *   brainpoolP384t1: RFC 5639 curve over a 384 bit prime field
64      brainpoolP512r1: RFC 5639 curve over a 512 bit prime field
65      brainpoolP512t1: RFC 5639 curve over a 512 bit prime field
66      SM2       : SM2 curve over a 256 bit prime field
```

A criptografia de curvas elípticas ainda é uma área de pesquisa bastante ativa na qual existe uma grande variedade de curvas elípticas propostas para usos criptográficos, ao mesmo tempo em que há também esforços para comprovar ou refutar as alegações de segurança feitas para determinadas curvas. Por isto, é importante observar as fontes supracitadas em busca de atualizações de segurança.

5.4.15. Assinaturas digitais com ECDSA

Esta seção mostra os comandos `OpenSSL` usados para selecionar curvas elípticas, gerar pares de chaves criptográficas a partir das curvas selecionadas, gerar assinaturas digitais com o algoritmo ECDSA e verificar estas assinaturas.

Na Listagem 5.2, os comandos `ecparam -name`, nas linhas 1 e 2, criam parâmetros criptográficos para as duas curvas elípticas secp521r1 e prime256v1, respectivamente, e gravam-nos em arquivos de saída. Já os dois comandos `ecparam -genkey` nas linhas 4 e 5 geram os pares de chaves criptográficas para as duas curvas selecionadas anteriormente. Nas linhas 7 e 11, as chaves públicas dos dois pares de chaves são exportadas em arquivos separados pelos comandos `ec -pubout`.

Além disso, nas linhas 15 e 16, o comando `dgst` é usado para gerar duas assinaturas digitais sobre o *hash* (opção `-sha512`) do mesmo arquivo de entrada `gabarito.txt`, resultando em uma assinatura para cada curva selecionada neste exemplo. O comando `dgst`

`-sha512 -verify` da linha 18 verifica a assinatura digital gerada com a curva secp512r1, enquanto o comando `dgst -sha512 -verify` análogo, na linha 21, verifica a assinatura digital gerada com a curva prime256v1. As duas verificações são bem-sucedidas.

Finalmente, os comandos `ecparam` das linhas 24 e 28 mostram duas maneiras diferentes de descobrir informações de uma curva a partir dos parâmetros. A primeira lista os nomes da curva. A segunda, mais detalhada, lista em detalhe os parâmetros da curva em hexadecimal: primo, *a*, *b*, gerador, ordem, cofator e semente.

Listagem 5.15. Assinaturas digitais e verificação de assinaturas com ECDSA.

```
1 C:\>openssl ecparam -name  secp521r1 -out ecparam1.pem
2 C:\>openssl ecparam -name prime256v1 -out ecparam2.pem
3
4 C:\>openssl ecparam -genkey -in ecparam1.pem -noout -out eckeys1.pem
5 C:\>openssl ecparam -genkey -in ecparam2.pem -noout -out eckeys2.pem
6
7 C:\>openssl ec -in eckeys1.pem -pubout -out public1.pem
8 read EC key
9 writing EC key
10
11 C:\>openssl ec -in eckeys2.pem -pubout -out public2.pem
12 read EC key
13 writing EC key
14
15 C:\>openssl dgst -SHA512 -sign eckeys1.pem -out signature1.sign gabarito.txt
16 C:\>openssl dgst -SHA512 -sign eckeys2.pem -out signature2.sign gabarito.txt
17
18 C:\>openssl dgst -SHA512 -verify public1.pem -signature signature1.sign gabarito.txt
19 Verified OK
20
21 C:\>openssl dgst -SHA512 -verify public2.pem -signature signature2.sign gabarito.txt
22 Verified OK
23
24 C:\>openssl ecparam -in ecparam2.pem -text -noout
25 ASN1 OID: prime256v1
26 NIST CURVE: P-256
27
28 C:\>openssl ecparam -in ecparam2.pem -text -param_enc explicit -noout
29 Field Type: prime-field
30 Prime:
31     00:ff:ff:ff:ff:00:00:00:01:00:00:00:00:00:00:
32     00:00:00:00:00:00:ff:ff:ff:ff:ff:ff:ff:ff:ff:
33     ff:ff:ff
34 A:
35     00:ff:ff:ff:ff:00:00:00:01:00:00:00:00:00:00:
36     00:00:00:00:00:00:ff:ff:ff:ff:ff:ff:ff:ff:ff:
37     ff:ff:fc
38 B:
39     5a:c6:35:d8:aa:3a:93:e7:b3:eb:bd:55:76:98:86:
40     bc:65:1d:06:b0:cc:53:b0:f6:3b:ce:3c:3e:27:d2:
41     60:4b
42 Generator (uncompressed):
43     04:6b:17:d1:f2:e1:2c:42:47:f8:bc:e6:e5:63:a4:
44     40:f2:77:03:7d:81:2d:eb:33:a0:f4:a1:39:45:d8:
45     98:c2:96:4f:e3:42:e2:fe:1a:7f:9b:8e:e7:eb:4a:
46     7c:0f:9e:16:2b:ce:33:57:6b:31:5e:ce:cb:b6:40:
47     68:37:bf:51:f5
48 Order:
49     00:ff:ff:ff:ff:00:00:00:00:ff:ff:ff:ff:ff:ff:
50     ff:ff:bc:e6:fa:ad:a7:17:9e:84:f3:b9:ca:c2:fc:
51     63:25:51
52 Cofactor:  1 (0x1)
53 Seed:
54     c4:9d:36:08:86:e7:04:93:6a:66:78:e1:13:9d:26:
55     b7:81:9f:7e:90
```

5.5. Avaliação de segurança criptográfica do TLS

Esta seção é organizada em torno de avaliações de segurança do SSL/TLS para detecção de suas vulnerabilidades nos servidores web por meio de comandos do `OpenSSL`, assim como também por outras ferramentas de análise de segurança, tais como, por exemplo: o `sslscan` [rbsec 2019], o `TestSSLServer` [Pornin 2017], o `SSLlabs` [Qualys 2019] e o `Observatory` [mozilla 2019], entre outras. Nesta seção, o `OpenSSL` é usado como uma ferramenta de testes e não como uma infraestrutura criptográfica, contrastando com a seção anterior.

5.5.1. Testando uma conexão TLS com `OpenSSL`

O `OpenSSL` vem com uma ferramenta cliente TLS que pode ser usada para estabelecer conexões com um servidor. O primeiro comando na Listagem 5.16 mostra como o cliente `s_client` usa a opção `-connect`, o nome do servidor e a porta 443 (padrão do SSL) para estabelecer uma conexão segura (na versão 1.2 `-tls1.2`) com um servidor web.

O suporte aos algoritmos criptográficos específicos pode ser testado pela opção `-cipher` do comando `s_client` seguida do nome da *suite* criptográfica. Já o comando `ciphers -s` lista os algoritmos suportados na instalação local no `OpenSSL`. Esta lista é ligeiramente diferente daquela mostrada por outras opções do comando `ciphers` analisadas adiante no texto.

Os outros comandos na Listagem 5.16 mostram os algoritmos disponíveis e depois testam a conexão com criptografias específicas: `-cipher AES128-SHA256` e `-cipher kECDHE`, que valida o uso do ECDH efêmero no acordo de chaves.

Se o endereço `exemplo.com` for substituído por uma URL verdadeira, o comando `s_client` emite informações diagnósticas da conexão com o servidor. O `s_client` espera que um comando HTTP seja fornecido para o servidor web. Uma maneira simples de fazer isto é digitar no *prompt* do terminal o comando HTTP `HEAD / HTTP/1.0` seguido opcionalmente da URL do servidor.

As informações de diagnóstico incluem dados da cadeia de certificação, o certificado do servidor, a versão do protocolo, a *suite* criptográfica, o tamanho da chave pública do servidor, o estado das flags de renegociação e de compressão e várias informações da sessão SSL.

Listagem 5.16. O `OpenSSL` como um cliente TLS.

```
1 C:\>openssl s_client -connect exemplo.com:443 -tls1_2
2
3 C:\>openssl ciphers -s
4 TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA256:
5 ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384:
6 ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-CHACHA20-POLY1305:
7 ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES128-GCM-SHA256:
8 ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA384:DHE-RSA-AES256-SHA256:
9 ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA256:
10 ECDHE-ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA:
11 ECDHE-RSA-AES128-SHA:DHE-RSA-AES128-SHA:AES256-GCM-SHA384:AES128-GCM-SHA256:
12 AES256-SHA256:AES128-SHA256:AES256-SHA:AES128-SHA
13
14 C:\>openssl s_client -connect exemplo.com -port 443 -cipher AES128-SHA256
15
16 C:\>openssl s_client -connect exemplo.com -port 443 -cipher kECDHE
```

5.5.2. Testando o suporte do servidor às versões do SSL/TLS

O comportamento padrão do comando `s_client` é tentar estabelecer uma conexão na versão mais alta do protocolo disponível na instalação do `OpenSSL`. Versões específicas podem ser usadas explicitamente com as opções `-ssl2` (disponível apenas em versões antigas do `OpenSSL`), `-ssl3`, `-tls1`, `-tls1_1`, `-tls1_2` e `-tls1_3` (somente em distribuições novas). Além disso, versões do TLS podem ser excluídas explicitamente com as opções `-no_ssl2`, `-no_ssl3`, `-no_tls1`, `-no_tls1_1`, `-no_tls1_2`.

A Listagem 5.17 mostra o resultado da tentativa mal-sucedida do cliente `s_client -connect` em estabelecer uma conexão com um servidor web na versão 1.3 do TLS (opção `-tls1_3`). Este falha de conexão tende a se tornar menos comum à medida que mais servidores adotem a nova versão.

Listagem 5.17. Exemplo de servidor web sem suporte ao TLS 1.3.

```
C:\>openssl s_client -connect www.exemplo.com -port 443 -tls1_3
CONNECTED(000000E8)
5140:error:14094410:SSL routines:ssl3_read_bytes:sslv3 alert handshake failure:s
sl\record\rec_layer_s3.c:1528:SSL alert number 40
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 7 bytes and written 237 bytes
Verification: OK
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
```

O comando `ciphers -v "ALL"` (sem filtro) mostra em mais detalhe uma lista longa das *suites* criptográficas do SSL/TLS, que pode ser filtrada para versões ou algoritmos específicos. A Listagem 5.18 mostra o resultado do comando `ciphers -v "ALL"` filtrado apenas para as *suites* do TLSv1 que contém o acordo de chaves ECDH.

Os itens desta lista podem ser testados contra um servidor web com o comando `s_client -connect`, conforme exemplificado ao final da Listagem 5.18.

Listagem 5.18. Testes por *suites* criptográficas específicas.

```
C:\>openssl ciphers -v "ALL" | find "TLSv1 " | find "ECDH"
ECDHE-ECDSA-AES256-SHA   TLSv1 Kx=ECDH      Au=ECDSA Enc=AES(256)   Mac=SHA1
ECDHE-RSA-AES256-SHA     TLSv1 Kx=ECDH      Au=RSA   Enc=AES(256)   Mac=SHA1
AECDH-AES256-SHA         TLSv1 Kx=ECDH      Au=None  Enc=AES(256)   Mac=SHA1
ECDHE-ECDSA-AES128-SHA   TLSv1 Kx=ECDH      Au=ECDSA Enc=AES(128)   Mac=SHA1
ECDHE-RSA-AES128-SHA     TLSv1 Kx=ECDH      Au=RSA   Enc=AES(128)   Mac=SHA1
AECDH-AES128-SHA         TLSv1 Kx=ECDH      Au=None  Enc=AES(128)   Mac=SHA1
ECDHE-PSK-AES256-CBC-SHA384 TLSv1 Kx=ECDHEPSK Au=PSK  Enc=AES(256)   Mac=SHA384
ECDHE-PSK-AES256-CBC-SHA TLSv1 Kx=ECDHEPSK Au=PSK   Enc=AES(256)   Mac=SHA1
ECDHE-PSK-CAMELLIA256-SHA384 TLSv1 Kx=ECDHEPSK Au=PSK  Enc=Camellia(256) Mac=SHA384
ECDHE-PSK-AES128-CBC-SHA256 TLSv1 Kx=ECDHEPSK Au=PSK   Enc=AES(128)   Mac=SHA256
ECDHE-PSK-AES128-CBC-SHA TLSv1 Kx=ECDHEPSK Au=PSK   Enc=AES(128)   Mac=SHA1
ECDHE-PSK-CAMELLIA128-SHA256 TLSv1 Kx=ECDHEPSK Au=PSK  Enc=Camellia(128) Mac=SHA256

C:\>openssl s_client -connect www.exemplo.com:443 -cipher ECDHE-ECDSA-AES256-SHA
```

5.5.3. Reuso de conexões e renegociação segura

A capacidade de reconectar utilizando parâmetros já calculados anteriormente é importante na garantia de disponibilidade de um servidor SSL/TLS. Por outro lado, as conexões SSL/TLS entre cliente e servidor web mantém parâmetros que devem ser recalculados periodicamente, caso contrário a sua utilização por tempo indeterminado ou por uma quantidade grande de reconexões traz insegurança tanto para o cliente quanto para o servidor. Assim, há um compromisso entre a quantidade de reconexões e a insegurança de reutilização de parâmetros.

A Listagem 5.19 mostra como a opção `-reconnect` do comando `s_client` pode ser usada para determinar quantas reconexões um servidor web permite para uma conexão SSL/TLS. Conforme ilustrado na Listagem 5.19, cinco reusos são esperados antes de uma nova conexão (omitida no texto juntamente com o restante da saída).

Listagem 5.19. Teste de reuso de conexões.

```
1 C:\>openssl s_client -connect www.exemplo.com:443 -reconnect | find /I "Reuse"
2 depth=2 C = GB, ST = Greater Manchester
      , L = Salford, O = COMODO CA Limited, CN = COMODO RSA Certification Authority
3 verify error:num=20:unable to get local issuer certificate
4 Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
5 Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
6 Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
7 Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
8 Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
9 read:errno=0
```

A renegociação segura de parâmetros de conexão pode ser detectada facilmente nas versões novas do TLS. A Listagem 5.20 mostra o trecho da saída de um comando `s_client -connect` em que se pode observar a frase afirmativa *"Secure Renegotiation IS supported"* indicando o suporte a este característica. O contrário seria indicado explicitamente pela frase negativa.

Mesmo um servidor que suporta renegociação ainda pode rejeitar a renegociação iniciada pelo cliente, por que ela aumenta a superfície de ataque do servidor, abrindo oportunidade para ataques de *downgrade* sobre o protocolo. Dito isto, o `s_client -connect` pode solicitar a renegociação de parâmetros simplesmente digitando "R"na linha de comando do terminal (linhas 10 e 11). Se a renegociação for possível, o servidor reinicia o protocolo de *handshake* e envia novamente seu certificado. Se a renegociação iniciada pelo cliente não for possível, uma mensagem de erro (parecida com a mostrada nas linhas 13 e 14) é enviada pelo servidor.

Listagem 5.20. Renegociação segura.

```
1 C:\>openssl s_client -connect www.exemplo.com:443
2 [...]
3 New, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
4 Server public key is 2048 bit
5 Secure Renegotiation IS supported
6 Compression: NONE
7 Expansion: NONE
8 [...]
9
10 HEAD / HTTP/1.0
11 R
12 RENEGOTIATING
13 5620:error:1409444C:SSL routines:ssl3_read_bytes: tlsv1 alert no renegotiation:
14 ssl\record\rec_layer_s3.c:1528: SSL alert number 100
```

5.5.4. Verificação de certificados revogados em tempo real

A verificação de revogação de um certificado não precisa ser feita apenas por meio de CRLs distribuídas por ACs. Uma alternativa é um cliente SSL/TLS usar o *Online Certificate Status Protocol* (OCSP) para consultar a AC em tempo real sobre o *status* de revogação de um certificado.

Há desvantagens nesta abordagens. Primeiramente, se o serviço OCSP está indisponível por qualquer razão, o cliente poderá decidir por aceitar a conexão mesmo sem ter verificado a revogação do certificado do servidor. Segundo, a AC pode ficar sobrecarregada ao responder em tempo real para uma grande quantidade de clientes SSL/TLS, resultando não apenas em atrasos para o cliente, mas também em indisponibilidade do serviço OCSP. O método OCSP *Stapling* resolve estas duas questões.

O OCSP *Stapling* (grampeamento) permite que um servidor (por exemplo, HTTPS) anexe o seu *status* OCSP ao *handshake* do protocolo SSL/TLS. O *status* OCSP do servidor é requisitado pelo próprio servidor à AC emissora, periodicamente, e é assinado pela AC. A Listagem 5.21 mostra como a opção `-status` do comando `s_client` faz com que o cliente solicite explicitamente o *status* do servidor via OCSP *Stapling*. A Listagem 5.21 também mostra a resposta da AC grampeada à resposta do servidor. O restante da saída é omitido.

Listagem 5.21. Teste de grampeamento de OCSP (*OCSP Stapling*).

```
1 C:\>openssl s_client -connect www.exemplo.com:443 -status
2 CONNECTED(000000E8)
3 depth=1 C = US,
      O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert SHA2 High Assurance Server CA
4 verify error:num=20:unable to get local issuer certificate
5 OCSP response:
6 ======================================
7 OCSP Response Data:
8     OCSP Response Status: successful (0x0)
9     Response Type: Basic OCSP Response
10    Version: 1 (0x0)
11    Responder Id: 5168FF90AF0207753CCCD9656462A212B859723B
12    Produced At: Feb 19 06:26:58 2019 GMT
13    Responses:
14    Certificate ID:
15      Hash Algorithm: sha1
16      Issuer Name Hash: CF26F518FAC97E8F8CB342E01C2F6A109E8E5F0A
17      Issuer Key Hash: 5168FF90AF0207753CCCD9656462A212B859723B
18      Serial Number: 0FDC2551201743C7F56141EC293D66BD
19    Cert Status: good
20    This Update: Feb 19 06:26:58 2019 GMT
21    Next Update: Feb 26 05:41:58 2019 GMT
22
23    Signature Algorithm: sha256WithRSAEncryption
24         0c:54:dd:57:25:1b:55:60:e8:8e:81:29:89:76:4d:8c:9c:f7:
25         33:e4:c9:1a:9f:7e:ae:8b:df:f0:11:87:29:5d:bd:dc:01:5d:
26         87:c9:09:83:f3:c5:b0:d5:4f:68:07:7d:25:7a:54:76:5f:98:
27         d8:af:6e:1e:86:bc:73:6b:20:5d:3f:2e:ad:b8:00:3e:c9:b2:
28         00:db:2f:86:34:24:7a:34:5f:a2:27:27:41:f9:52:6e:bb:b0:
29         07:8a:3e:88:a6:8f:94:61:db:65:31:76:7d:cf:70:91:c2:58:
30         57:c6:34:bf:27:31:2b:2f:3b:6f:84:57:bf:0a:07:42:b2:1b:
31         a2:15:25:e0:35:09:f6:8c:7f:a9:d5:ff:87:d2:88:45:50:46:
32         fb:01:fd:09:da:f6:3a:2c:0a:38:ff:0f:5b:03:9d:77:56:5b:
33         d7:56:63:4a:6e:07:cd:fd:84:f8:0c:af:3f:14:1b:e9:f7:17:
34         d6:cb:c2:d0:3d:0a:99:2d:97:a5:72:15:f2:1e:fe:9d:45:f7:
35         f7:86:7e:56:9e:4b:9a:9a:41:86:f5:f3:9a:43:1e:8a:61:2d:
36         10:4e:cc:96:f1:48:f9:03:f6:c0:2d:a2:62:ca:4c:e9:72:2f:
37         ad:b2:dc:43:f1:d1:84:19:19:48:8c:7c:b3:73:12:cb:0f:66:
38         14:3b:ba:e1
39 ======================================
```

5.5.5. Teste contra o ataque BEAST

O ataque *Browser Exploit Against SSL/TLS* (BEAST) é relativamente antigo (2011) e só afeta principalmente os clientes SSL/TLS nas versões até o TLSv1.0 [Ristic 2011]. Os testes de verificação contra o ataque BEAST no lado servidor ajudam a entender diversas boas práticas de segurança criptográficas implementadas nos servidores web, como, por exemplo, evitar o uso do modo de operação CBC e da encriptação de fluxo com RC4.

O RC4 foi considerado uma alternativa para o CBC no TLSv1.0 na época em que não havia outra encriptação de fluxo disponível para ser usada em vez do RC4. Esta restrição do protocolo levou o público em geral a acreditar que o RC4 era uma solução geral para o BEAST, resultando em substituições inseguras do CBC pelo RC4 em protocolos proprietários [Alkemade 2013]. Por exemplo, o mau uso do RC4 em um protocolo de comunicação segura permitiu a decriptação de mensagens encriptadas trocadas entre dois usuários de um aplicativo de comunicação instantânea [Alkemade 2013]. A vulnerabilidade consistiu na reutilização do fluxo de chaves para encriptar mensagens nas duas direções do canal de comunicação protegido pelo encriptador de fluxo.

Atualmente, a estratégia administrativa recomendada para mitigação do BEAST é adotar apenas as versões mais novas do TLS (1.1, 1.2 e 1.3). Porém, naquelas situações em que é necessário manter o TLSv1.0 para compatibilidade com o legado, apenas uma mitigação parcial é possível. A Listagem 5.22 mostra dois comandos `s_client -connect`. O comando da linha 1 inibe o RC4 e, por isto, só estabelece a conexão se houver alguma criptografia simétrica com CBC para ser usada no TLSv1.0 ou mais baixa. Isto pode ser observado nas linhas 4, 11 e 12. Já o comando da linha 15 oferece o RC4 caso o TLSv1.0 priorize esta mitigação. Como pode ser observado nas linhas 18, 25 e 26, o servidor estabeleceu a conexão vulnerável, indicando que não há mitigação implantada contra o BEAST.

Listagem 5.22. Teste de BEAST.

```
1 C:\>openssl s_client
      -connect exemplo.com:443 -cipher "ALL:!RC4" -no_tls1_1 -no_tls1_2 -no_tls1_3
2
3 [...]
4 New, TLSv1.0, Cipher is ECDHE-RSA-AES128-SHA
5 Server public key is 2048 bit
6 Secure Renegotiation IS supported
7 Compression: NONE
8 Expansion: NONE
9 No ALPN negotiated
10 SSL-Session:
11     Protocol  : TLSv1
12     Cipher    : ECDHE-RSA-AES128-SHA
13 [...]
14
15 C:\>openssl s_client
      -connect exemplo.com:443 -cipher "ALL:+RC4" -no_tls1_1 -no_tls1_2 -no_tls1_3
16
17 [...]
18 New, TLSv1.0, Cipher is ECDHE-RSA-AES128-SHA
19 Server public key is 2048 bit
20 Secure Renegotiation IS supported
21 Compression: NONE
22 Expansion: NONE
23 No ALPN negotiated
24 SSL-Session:
25     Protocol  : TLSv1
26     Cipher    : ECDHE-RSA-AES128-SHA
27 [...]
```

5.5.6. Teste de parâmetros do DH/ECDH

Em versões recentes do `OpenSSL`, o comando `s_client` mostra o tamanho dos parâmetros DH/ECDH usados na conexão recém estabelecida. A Listagem 5.23 mostra três comandos `s_client` com opções `-cipher` diferentes que resultam em parâmetros criptográficos diferentes em cada caso. Nos três casos, as informações de saída localizadas antes e depois dos pontos de interesse foram omitidas por questões de espaço. Estas configurações são suficientemente seguras contra ataques recentes como o LogJam [Log 2016].

No primeiro caso mostrado nas linhas de 1 a 9 da Listagem 5.23, as assinaturas digitais das mensagens do protocolo são geradas com ECDSA sobre SHA256 e as chaves criptográficas foram geradas com tamanho de 253 bits sobre a curva X25519. Já no segundo caso mostrado nas linhas de 11 a 19, as assinaturas digitais das mensagens do protocolo são geradas com RSA sobre SHA512 e as chaves criptográficas ECDH foram geradas com tamanho de 256 bits sobre a curva NIST P-256.

Quando ECDH é utilizado, é importante observar o uso de curvas elípticas fortes e seguras, conforme descrito anteriormente, além de chaves criptográficas suficientemente grandes, com nível de segurança de no mínimo 128 bits (256 bits de tamanho), conforme tabela 5.1. Assim, as curvas X25519 e NIST P-256 são escolhas boas.

Finalmente, no terceiro caso mostrado nas linhas de 21 até 29 da Listagem 5.23, as assinaturas digitais das mensagens do protocolo são geradas com RSA sobre SHA512 e as chaves criptográficas DH foram geradas com tamanho de 2048 bits. Neste caso, vale observar que chaves DH menores de 2048 bits (por exemplo, 512 ou 1024 bits) já são consideradas inseguras para os padrões atuais por que favorecem o ataque LogJam [Log 2016].

Listagem 5.23. Teste de parâmetros de DH/ECDH.

```
1 C:\>openssl s_client -connect www.exemplo.com:443 -cipher ECDHE-ECDSA-AES256-SHA
2 [...]
3 ---
4 No client certificate CA names sent
5 Peer signing digest: SHA256
6 Peer signature type: ECDSA
7 Server Temp Key: X25519, 253 bits
8 ---
9 [...]
10
11 C:\>openssl s_client -connect exemplo.com -port 443 -cipher kECDHE
12 [...]
13 ---
14 No client certificate CA names sent
15 Peer signing digest: SHA512
16 Peer signature type: RSA
17 Server Temp Key: ECDH, P-256, 256 bits
18 ---
19 [...]
20
21 C:\>openssl s_client -connect exemplo.com -port 443 -cipher kDHE
22 [...]
23 ---
24 No client certificate CA names sent
25 Peer signing digest: SHA512
26 Peer signature type: RSA
27 Server Temp Key: DH, 2048 bits
28 ---
29 [...]
```

5.5.7. Testes automático de SSL/TLS com SSLscan

A ferramenta `SSLscan` [rbsec 2019], como toda ferramenta autônoma de varredura de vulnerabilidades, tem a vantagem de poder ser usada para testar um servidor SSL/TLS na rede interna (sem endereço IP público). A Listagem 5.24 mostra o comando `SSLscan` usando para varrer um servidor fictício, onde a opção `-verbose` indica que muitas informações são mostradas na saída. As linhas de 7 até 13 mostram o status de algumas configurações do protocolo, enquanto as linhas de 15 até 18 mostram quais versões são vulneráveis ou não ao Heartbleed [Synopsys 2014]. Em particular, a linha 11 mostra que há suporte à renegociação segura e a linha 13 mostra que a compressão está desabilitada como mitigação contra o ataque CRIME [CRI 2012].

Finalmente, as linhas de 20 até 40 listam as *suites* criptográficas suportadas pelo servidor nas versões TLSv1, TLSv1.1 e TLSv1.2 do protocolo, assim como também a curva elíptica utilizada e o tamanho da chave DHE. As *suites* nas linha 21, 33 e 37 são as preferidas pelo servidor. As informações do certificado foram omitidas.

Listagem 5.24. Ferramenta de teste SSLscan.

```
1 C:\>sslscan -verbose www.exemplo.com:443
2 Version: 1.11.11 Windows 64-bit (Mingw)
3 OpenSSL 1.0.2-chacha (1.0.2g-dev)
4
5 Connected to 74.6.144.137
6
7 Testing SSL server www.exemplo.com on port 443 using SNI name www.exemplo.com
8
9   TLS Fallback SCSV: Server supports TLS Fallback SCSV
10
11  TLS renegotiation: Secure session renegotiation supported
12
13  TLS Compression: Compression disabled
14
15  Heartbleed:
16 TLS 1.2 not vulnerable to heartbleed
17 TLS 1.1 not vulnerable to heartbleed
18 TLS 1.0 not vulnerable to heartbleed
19
20  Supported Server Cipher(s):
21 Preferred TLSv1.2  128 bits  ECDHE-RSA-AES128-GCM-SHA256   Curve P-256 DHE 256
22 Accepted  TLSv1.2  256 bits  ECDHE-RSA-AES256-GCM-SHA384   Curve P-256 DHE 256
23 Accepted  TLSv1.2  128 bits  ECDHE-RSA-AES128-SHA256       Curve P-256 DHE 256
24 Accepted  TLSv1.2  256 bits  ECDHE-RSA-AES256-SHA384       Curve P-256 DHE 256
25 Accepted  TLSv1.2  128 bits  ECDHE-RSA-AES128-SHA          Curve P-256 DHE 256
26 Accepted  TLSv1.2  256 bits  ECDHE-RSA-AES256-SHA          Curve P-256 DHE 256
27 Accepted  TLSv1.2  128 bits  AES128-GCM-SHA256
28 Accepted  TLSv1.2  256 bits  AES256-GCM-SHA384
29 Accepted  TLSv1.2  128 bits  AES128-SHA256
30 Accepted  TLSv1.2  256 bits  AES256-SHA256
31 Accepted  TLSv1.2  128 bits  AES128-SHA
32 Accepted  TLSv1.2  256 bits  AES256-SHA
33 Preferred TLSv1.1  128 bits  ECDHE-RSA-AES128-SHA          Curve P-256 DHE 256
34 Accepted  TLSv1.1  256 bits  ECDHE-RSA-AES256-SHA          Curve P-256 DHE 256
35 Accepted  TLSv1.1  128 bits  AES128-SHA
36 Accepted  TLSv1.1  256 bits  AES256-SHA
37 Preferred TLSv1.0  128 bits  ECDHE-RSA-AES128-SHA          Curve P-256 DHE 256
38 Accepted  TLSv1.0  256 bits  ECDHE-RSA-AES256-SHA          Curve P-256 DHE 256
39 Accepted  TLSv1.0  128 bits  AES128-SHA
40 Accepted  TLSv1.0  256 bits  AES256-SHA
41
42  SSL Certificate:
43 Signature Algorithm: sha256WithRSAEncryption
44 RSA Key Strength:    2048
45 [...certificado omitido ...]
```

5.5.8. Testes automáticos de SSL/TLS com TestSSLServer

A ferramenta `TestSSLServer` [Pornin 2017] também tem a vantagem de poder ser usada para testar um servidor SSL/TLS sem endereço IP público. A Listagem 5.25 mostra o comando `TestSSLServer`, onde a opção `-ec` indica que as configurações de curvas elípticas do servidor são mostradas. As linhas de 4 até 10 mostram as *suites* criptográficas preferidas pelo servidor para as versões TLSv1 e TLSv1.1 do protocolo, enquanto as linhas de 11 até 24 mostram as *suites* da versão TLSv1.2 preferidas, onde se pode observar *suites* sem DH/ECDH, apenas com RSA, que não oferecem *forward secrecy*, conforme aviso da linha 51. As informações da cadeia de certificação foram omitidas. As linhas de 28 até 35 mostram o *status* de diversas configurações, enquanto as linhas de 35 até 49 mostram as curvas elípticas suportadas pelo servidor.

Listagem 5.25. Ferramentas de teste TestSSLServer.

```
1 C:\>TestSSLServer2.exe -ec  www.exemplo.com 443
2 Connection: www.exemplo.com:443
3 SNI: www.exemplo.com
4   TLSv1.0:
5       server selection: enforce server preferences
6       3f- (key:  RSA)  ECDHE_RSA_WITH_AES_128_CBC_SHA
7       3f- (key:  RSA)  ECDHE_RSA_WITH_AES_256_CBC_SHA
8       3-- (key:  RSA)  RSA_WITH_AES_128_CBC_SHA
9       3-- (key:  RSA)  RSA_WITH_AES_256_CBC_SHA
10  TLSv1.1: idem
11  TLSv1.2:
12      server selection: enforce server preferences
13      3f- (key:  RSA)  ECDHE_RSA_WITH_AES_128_GCM_SHA256
14      3f- (key:  RSA)  ECDHE_RSA_WITH_AES_256_GCM_SHA384
15      3f- (key:  RSA)  ECDHE_RSA_WITH_AES_128_CBC_SHA256
16      3f- (key:  RSA)  ECDHE_RSA_WITH_AES_256_CBC_SHA384
17      3f- (key:  RSA)  ECDHE_RSA_WITH_AES_128_CBC_SHA
18      3f- (key:  RSA)  ECDHE_RSA_WITH_AES_256_CBC_SHA
19      3-- (key:  RSA)  RSA_WITH_AES_128_GCM_SHA256
20      3-- (key:  RSA)  RSA_WITH_AES_256_GCM_SHA384
21      3-- (key:  RSA)  RSA_WITH_AES_128_CBC_SHA256
22      3-- (key:  RSA)  RSA_WITH_AES_256_CBC_SHA256
23      3-- (key:  RSA)  RSA_WITH_AES_128_CBC_SHA
24      3-- (key:  RSA)  RSA_WITH_AES_256_CBC_SHA
25 ========================================
26 [... cadeia de certificados omitida ... ]
27 ========================================
28 Server compression support: no
29 Server sends a random system time.
30 Secure renegotiation support: yes
31 Encrypt-then-MAC support (RFC 7366): no
32 SSLv2 ClientHello format (for SSLv3+): yes
33 Minimum EC size (no extension):   256
34 Minimum EC size (with extension): 256
35 ECDH parameter reuse:  no
36 Supported curves (size and name) ('*' = selected by server):
37      281  sect283k1 (K-283)
38      282  sect283r1 (B-283)
39      407  sect409k1 (K-409)
40      408  sect409r1 (B-409)
41      569  sect571k1 (K-571)
42      570  sect571r1 (B-571)
43      256  secp256k1
44    * 256  secp256r1 (P-256)
45      384  secp384r1 (P-384)
46      521  secp521r1 (P-521)
47      256  brainpoolP256r1
48      384  brainpoolP384r1
49      512  brainpoolP512r1
50 ========================================
51 WARN[CS006]: Server supports cipher suites with no forward secrecy.
```

5.6. Robustecimento e configuração segura do TLS

Esta seção trata as práticas de configuração segura do protocolo SSL/TLS nos servidores web Apache por meio do robustecimento das configurações do módulo `mod_ssl` [mod_ssl 2019]. A seção também cobre a sintaxe de nomenclatura utilizada para descrição de *suítes* criptográficas no SSL/TLS. O módulo `mod_ssl` é a interface do servidor web Apache para o `OpenSSL`. O site oficial [mod_ssl 2019] tem instruções para *download* e instalação do `mod_ssl` e a sua configuração no Apache. Este texto supõe que o `mod_ssl` já está instalado no servidor Apache, tratando apenas da configuração do módulo.

Aplicando técnicas de auditoria de sistemas, a utilização correta das configurações seguras do SSL/TLS pode ser verificada por meio de inspeções das configurações do `mod_ssl` do servidor web em execução, complementando as técnicas de teste tratadas na seção anterior. Por exemplo, um auditor de sistemas é capaz de verificar a presença do módulo `mod_ssl` em um servidor web Apache ativo (em execução) por meio do comando `httpd.exe -M`, que lista todos os módulos instalados e carregados pelo servidor web. Já que podem existir muitos módulos, um filtro simples seleciona e mostra apenas a linha do `mod_ssl`, se ela existir, como visto a seguir.

```
C:\>httpd.exe -M | find "ssl"
 ssl_module (shared)
```

5.6.1. As *suites* criptográficas das versões TLSv1.2 e TLSv1.3

Uma *suite* criptográfica é um conjunto de funções ou rotinas de criptografia utilizado em uma instância de comunicação segura com o protocolo SSL/TLS. A nomenclatura das *suites* criptográficas segue regras bem definidas que identificam todas as rotinas criptográficas necessárias em um canal de comunicação segura com SSL/TLS. Por exemplo, a Listagem 5.26 mostra algumas *suites* criptográficas disponíveis para o TLsv1.2 que contém ECDHE e AES (linhas de 1 até 13), assim como também para o TLSv1.3 (linhas de 15 até 18), onde, neste último caso, as *suites* são exclusivas da versão 1.3 do protocolo.

Listagem 5.26. Exemplos de *suites* criptográficas nas versões 1.2 e 1.3 do TLS.

```
 1 C:\>openssl ciphers -v "TLSv1.2"  | find "ECDHE" | find "AES"
 2 ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESGCM(256)     Mac=AEAD
 3 ECDHE-RSA-AES256-GCM-SHA384   TLSv1.2  Kx=ECDH   Au=RSA    Enc=AESGCM(256)     Mac=AEAD
 4 ECDHE-ECDSA-AES256-CCM8       TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESCCM8(256)    Mac=AEAD
 5 ECDHE-ECDSA-AES256-CCM        TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESCCM(256)     Mac=AEAD
 6 ECDHE-ECDSA-AES128-GCM-SHA256 TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESGCM(128)     Mac=AEAD
 7 ECDHE-RSA-AES128-GCM-SHA256   TLSv1.2  Kx=ECDH   Au=RSA    Enc=AESGCM(128)     Mac=AEAD
 8 ECDHE-ECDSA-AES128-CCM8       TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESCCM8(128)    Mac=AEAD
 9 ECDHE-ECDSA-AES128-CCM        TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESCCM(128)     Mac=AEAD
10 ECDHE-ECDSA-AES256-SHA384     TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AES(256)        Mac=SHA384
11 ECDHE-RSA-AES256-SHA384       TLSv1.2  Kx=ECDH   Au=RSA    Enc=AES(256)        Mac=SHA384
12 ECDHE-ECDSA-AES128-SHA256     TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AES(128)        Mac=SHA256
13 ECDHE-RSA-AES128-SHA256       TLSv1.2  Kx=ECDH   Au=RSA    Enc=AES(128)        Mac=SHA256
14
15 C:\>openssl ciphers -v "TLSv1.3"
16 TLS_AES_256_GCM_SHA384        TLSv1.3 Kx=any Au=any Enc=AESGCM(256)             Mac=AEAD
17 TLS_CHACHA20_POLY1305_SHA256  TLSv1.3 Kx=any Au=any Enc=CHACHA20/POLY1305(256) Mac=AEAD
18 TLS_AES_128_GCM_SHA256        TLSv1.3 Kx=any Au=any Enc=AESGCM(128)             Mac=AEAD
```

Na Listagem 5.26, o comando `openssl ciphers -v "TLSv1.2"` da linha 1 mostra doze (12) *suites* criptográficas da versão TLsv1.2 que usam o acordo de chaves ECDHE

(Diffie-Helman efêmero sobre curvas elípticas) e AES para encriptação simétrica. A análise de cada uma das *suites* desta lista permite identificar os elementos de construção, da esquerda para a direita:

- Criptografia assimétrica para distribuição de chaves, indicando um mecanismo de acordo de chaves ou de transporte de chaves (no exemplo, apenas o ECDHE);
- Esquema de assinaturas digitais para garantia de autenticidade das mensagens do protocolo de *handshake* (no exemplo, RSA PKCS#1v1.5 e ECDSA);
- Encriptação simétrica usada na proteção da confidencialidade dos dados, indicando o nome do algoritmo, o tamanho da chave de sessão e o modo de operação (no exemplo, AES-GCM, AES-CCM, AES-CCM8 ou AES-CBC);
- Mecanismo de MAC ou função de *hash* que compõe o cálculo do HMAC usado na autenticação das mensagens de troca de dados (no exemplo, AEAD, SHA384 ou SHA256).

Já o comando `openssl ciphers -v "TLSv1.3"` da linha 15 mostra apenas três *suites* criptográficas exclusivas para a versão TLSv1.3. Isto ocorre porque nesta versão do protocolo, as opções de criptografia assimétrica para distribuição de chaves (p.ex., ECDHE) e para assinaturas digitais (p.ex., ECDSA) podem ser escolhidas pelo software cliente já no início do *handshake*. Restando, então, apenas a necessidade de negociar com o servidor o mecanismo de encriptação autenticada, com as opções: AES-GCM-128, AES-GCM-256 e CHACHA20-POLY1305-SHA256.

Uma seleção de *suites* criptográficas pode ser escolhida com o uso de palavras chave combinadas com os nomes dos algoritmos criptográficos, formando *strings* de configuração utilizadas por diversas aplicações que incorporam o `OpenSSL`, como é o caso do mod_ssl. Por exemplo, as seguintes *keywords* podem ser usadas tanto no comando `openssl ciphers -v`, quanto na diretiva de configuração `SSLCipherSuite` do módulo `mod_ssl`:

- "DEFAULT":*suites default* da instalação;
- "ALL": todas as *suites*, exceto aquelas com encriptação nula;
- "COMPLEMENTOFDEFAULT": "ALL"menos "DEFAULT";
- "COMPLEMENTOFALL: as *suites* com encriptação nula;
- "HIGH": *suites* com nível de segurança maior que 128 bits;
- "MEDIUM": *suites* com nível de segurança de 128 bits;
- "LOW": *suites* com nível de segurança menor que 128 bits (somente em versões antigas);
- "SSLv3", "TLSv1", "TLSv1.1TLSv1.2", "TLSv1.3": todas as *suites* da versão;
- "SHA", "SHA1", "SHA256", "SHA384": *suites* que usam estes *hashes*;
- "aDH", "aDSS", "aECDH", "aECDSA", "aRSA": *suites* com autenticação;
- "aNULL": *suites* sem autenticação (p.ex., DH anônimo);
- "PSK", "SRP": *suites* com autenticação por chave predefinida ou por senha segura;
- "ADH","AECDH": *suites* inseguras que usam DH anônimo;

- "DH","ECDH": *suites* que usam qualquer DH (anônimo ou efêmero);
- "EDH", "EECDH": *suites* inseguras que usam DH efêmero;
- "kEDH", "kEECDH": *suites* que usam DH efêmero e anônimo;
- "RSA, "kRSA": *suites* que fazem troca de chaves com RSA;
- "AES","AESGCM": *suites* com AES simples ou encriptação autenticada GCM;
- "eNULL","NULL": *suites* sem encriptação.

A *string* de configuração de uma *suite* criptográfica personalizada pode ser montada pela combinação de *keywords*, da esquerda para a direita, separadas por dois pontos (":") ou espaço. O comportamento *default* é sempre adicionar ao final da lista as *suites* associadas à *keyword* a direita. Por exemplo, o comando `openssl ciphers -v "aNULL:eNULL"` seleciona as *suites* sem autenticação e acrescenta à lista as *suites* sem encriptação.

Este comportamento de adicionar *suites* ao final da lista pode ser modificado das seguintes maneiras: um sinal de menos (-") antes da *keyword* elimina as *suites* da lista se elas estiveram lá, mas elas ainda podem ser adicionadas novamente por outra *keyword*, que apareça depois, mais a direita. Já um sinal de exclamação ("!") remove a *suite* definitivamente, enquanto o sinal de mais ("+") move para o final da lista uma *suite* que já está na *string* de configuração. Finalmente, a *keyword* "@STRENGTH"ordena as *suites* da lista pela ordem decrescente do nível de segurança (isto é, a mais segura aparece primeiro, a lista vai da mais segura para a menos segura).

5.6.2. As configurações criptográficas seguras do `mod_ssl`

De acordo com a documentação [Apache Software Foundation 2019], as configurações do servidor Apache (a partir da versão 2.4) estão localizadas nos arquivos `httpd.conf` e `httpd-ssl.conf`. Vale a pena observar a documentação específica da versão utilizada do Apache a fim de determinar a localização dos arquivos de configuração e eventuais mudanças de nomenclatura. O Apache oferece um número grande de configurações em vários aspectos. Este texto trata apenas das configurações criptográficas relacionadas ao `mod_ssl`. Em particular, esta subseção mostra as configurações do `mod_ssl` que utilizam ou afetam a seleção das *suites* criptográficas. Além disso, são analisadas as configurações seguras do `mod_ssl` seguindo boas práticas atualmente em uso [mozilla 2018, SSL Labs 2019]. Primeiramente, são verificadas as configurações que determinam que o módulo foi carregado.

Listagem 5.27. Configurações mínimas do `mod_ssl` no `httpd.conf` do Apache 2.4.x.

```
1 ...
2
3 LoadModule ssl_module modules/mod_ssl.so
4 ...
5
6 # Secure (SSL/TLS) connections
7 Include conf/extra/httpd-ssl.conf
8 ...
9
10 <IfModule ssl_module>
11 SSLRandomSeed startup builtin
12 SSLRandomSeed connect builtin
13 </IfModule>
14 ...
```

A configuração inicial mínima do `mod_ssl` no Apache consiste das seguintes diretivas presentes no arquivo de configuração `httpd.conf` mostradas na Listagem 5.27. As reticências indicam que o arquivo de configuração é longo e muitas linhas foram omitidas. A linha 3 mostra a configuração `LoadModule` para a carga do módulo. Na linha 7, a diretiva `Include` indica que as configurações específicas do módulo estão em um aquivo de configuração separado (`httpd-ssl.conf`) em uma pasta extra. Finalmente, as configurações das linhas de 10 até 13 indicam que o SSL/TLS sempre utilizará sementes pseudo-aleatórias novas tanto na sua inicialização, quanto no estabelecimento de conexões.

A Listagem 5.28 mostra as configurações presentes no arquivo `httpd-ssl.conf` conforme explicado a seguir. A configuração `Listen 443`, na linha 3, indica que o protocolo HTTPS usará a porta 443 (de fato, a porta *default* deste protocolo). As configurações `SSLCipherSuite` e `SSLProxyCipherSuite`, nas linhas 7 e 8, indicam as *suites* criptográficas que serão negociadas pelo servidor Apache diretamente com o software cliente ou por meio de um *proxy* de conexões. A *string* de configuração é uma lista que pode conter *suites* criptográficas, nomes dos algoritmos e as *keywords* explicadas anteriormente. No exemplo da linha 7, primeiro são incluídas as *suites* com nível de segurança maior que 128 bits, depois aquelas com nível de segurança igual a 128 bits. Em seguida, são excluídas todas as *suites* que contenham os algoritmos MD5, RC4 e 3DES.

A configuração `SSLHonorCipherOrder on`, na linha 12, indica que a ordem em que as *suites* e algoritmos criptográficos aparecem na *string* de configuração `SSLCipherSuite` é importante e por isto deve ser honrada na negociação com o cliente SSL/TLS. Por exemplo, a ordem pode indicar que *suites* e algoritmos de melhor desempenho aparecem primeiro e foram escolhidos pelo administrador do servidor web da maneira indicada na *string* de configuração: primeiro as *suites* mais fortes e depois as menos fortes.

As configurações `SSLCompression off` e `SSLSessionTickets off`, nas linhas 16 e 17, indicam que os usos de compressão SSL/TLS e de *tickets* de sessão devem estar desligados. Estas são configurações antigas que, se ativadas, levam a vulnerabilidades conhecidas. Elas mantém a compatibilidade com versões antigas e sistemas legado e, na prática, ficam sempre desativadas em versões novas.

As configurações `SSLProtocol` e `SSLProxyProtocol`, nas linhas 21 e 22, informam quais versões do SSL/TLS são habilitadas no servidor. Geralmente, deseja-se todas as versões recentes, excluindo-se as versões antigas e inseguras (por exemplo, `All -SSLv2 -SSLv3 -TLSv1`). Versões mais recentes do `OpenSSL` já não oferecem a opção de configuração `-SSLv2`, uma vez que não possuem mais esta versão antiga do protocolo. Vale ainda observar que a configuração `SSLProtocol` mostrada segue o princípio da lista de exclusão (lista negativa). Outra opção é adotar uma lista de inclusão (lista positiva) que contenha explicitamente apenas as versões desejadas. Por exemplo, `TLSv1.1 TLSv1.2 TLSv1.3`).

As configurações relacionadas ao *cache* de sessão SSL/TLS, nas linhas 26 e 27, indicam o local de armazenamento e o tempo limite de duração da sessão. Já as configurações relacionadas ao OCSP *Stapling*, nas linhas de 31 até 34, indicam que o *Stapling* está ativado, onde o *cache* de OCSP está armazenado, e os tempos limite de erro e de permanência de uma resposta OSCP no *cache*.

No arquivo `httpd-ssl.conf`, o bloco ou contexto `<VirtualHost>` atende à

necessidade de compartimentação de aplicações no servidor Apache e contém configurações específicas das aplicações indicadas nos contextos. A Listagem 5.28 mostra o contexto *default* nas linhas de 38 até 49 que contém as configurações globais do servidor web (`<VirtualHost _default_:443>` ou `<VirtualHost *:443>`). Configurações específicas reproduzem este bloco de configuração e substituem o asterisco (*) pelo nome de domínio ou endereço IP do servidor virtual. Quando está presente no contexto `<VirtualHost>`, a configuração `SSLEngine On | Off` ativa ou desativa o SSL/TLS no contexto em questão. Já as configurações `SSLCertificateFile` e `SSLCertificateKeyFile` informam onde o Apache vai achar o certificado digital do virtual host e a chave privada correspondente.

Listagem 5.28. Configurações do `mod_ssl` no arquivo `httpd-ssl.conf` do Apache 2.4.

```
1
2 # qual a porta usada para o https
3 Listen 443
4 ...
5
6 # string configura as suites criptograficas com keywords e nomes do OpenSSL
7 SSLCipherSuite HIGH:MEDIUM:!MD5:!RC4:!3DES
8 SSLProxyCipherSuite HIGH:MEDIUM:!MD5:!RC4:!3DES
9 ...
10
11 # honra a ordem em que as suites sao listadas (motivo eh desempenho).
12 SSLHonorCipherOrder on
13 ...
14
15 # desabilta compression e session tickets
16 SSLCompression          off
17 SSLSessionTickets       off
18
19 ...
20 #permite TLsv1.1, TLSv1.2 e TLSv1.3 mas nao permite SSLv2, SSLv3 and TLsv1
21 SSLProtocol All -SSLv2 -SSLv3 -TLSv1
22 SSLProxyProtocol All -SSLv2 -SSLv3 -TLSv1
23 ...
24
25 #Cache da sessao (reutilizacao de parametros para favorecer desempenho)
26 SSLSessionCache         "shmcb:${SRVROOT}/logs/ssl_scache(512000)"
27 SSLSessionCacheTimeout  300
28 ...
29
30 # OCSP Stapling (desabilitado por default)
31 SSLUseStapling On
32 SSLStaplingCache "shmcb:${SRVROOT}/logs/ssl_stapling(32768)"
33 SSLStaplingStandardCacheTimeout 3600
34 SSLStaplingErrorCacheTimeout 600
35 ...
36
37 # Contexto do virtual host default
38 <VirtualHost _default_:443>
39   ...
40   # Habilita ou desabilita o SSL no virtual host
41   SSLEngine On
42
43   # Caminho para o certificado do servidor
44   SSLCertificateFile /usr/local/apache/conf/ssl/server.crt
45
46   # Caminho para a chave privada do servidor
47   SSLCertificateKeyFile /usr/local/apache/conf/ssl/server.key
48 ...
49 </VirtualHost>
50 ...
```

5.7. Considerações finais

Este texto descreve bons usos da criptografia no protocolo SSL/TLS, contribuindo para a administração de redes e de sistemas, facilitando o aprofundamento em estudos futuros. Em particular, profissionais atuando em redes de computadores e sistemas distribuídos são instruídos sobre SSL/TLS por meio de um tutorial `OpenSSL`, análises das configurações no `mod_ssl` e testes de segurança no servidor web Apache, O texto aborda a versão nova do SSL/TLS, ainda em processo de adoção pela indústria. Por isto, contribui para que administradores de TI entendam os benefícios da nova versão e avancem na atualização de seus sistemas.

Há oportunidades de pesquisa e desenvolvimento em SSL/TLS onde há escassez de especialistas em criptografia, e as configurações de SSL/TLS são feitas manualmente por profissionais de outras áreas, geralmente sobrecarregados com outras atividades. Três áreas em particular chamam a atenção: (*i*) a geração automática de configurações seguras, (*ii*) a verificação automática da segurança do TLS e (*iii*) a correção automática de configurações inseguras.

Profissionais de TI precisam de bons exemplos de configurações seguras para seguir atuando. Neste sentido, geradores de modelos de configuração seguras (atualizados para as versões novas e em diversas tecnologias) contribuem bastante nesta direção. Bons exemplos facilitam o aprendizado, servem de base para personalizações e aceleram as correções de defeitos. Por outro lado, maus exemplos (explicitamente documentados como tal e explicados) são uma ferramenta poderosa de conscientização sobre como evitar configurações inseguras.

A utilização de ferramentas de varredura de vulnerabilidades em implantações do SSL/-TLS é uma área já bastante ativa com uma variedade grande de ferramentas disponíveis. Os desafios tecnológicos desta área em particular estão não apenas na capacidade das ferramentas em diminuir a ocorrência de falsos positivos e de falsos negativos, mas principalmente na capacidade de comunicar suas descobertas de maneira adequada para não especialistas. Por isto, novas ferramentas de testes, fáceis de usar corretamente, atualizadas e amplamente disponíveis, também contribuem para a conscientização dos profissionais de TI em relação ao estado de suas infraestruturas, sendo um passo necessário na atualização de infraestruturas existentes.

Finalmente, no que se refere à detecção e correção automáticas de configurações inseguras, infraestruturas auto-adaptativas podem se beneficiar da detecção de configurações inseguras e da sua correção automática e transparente. Por exemplo, na implantação de contêineres de aplicação em infraestruturas de nuvem, um contêiner com um Apache configurado de modo inseguro para SSL/TLS poderia ser corrigido por edições controladas e automáticas de seus arquivos de configuração, em vez de ser rejeitado sumariamente.

Agradecimentos

Este trabalho foi realizado no Laboratório de Segurança e Criptografia (LASCA) do Instituto de Computação, Universidade Estadual de Campinas. Os autores agradecem o apoio financeiro do projeto ATMOSPHERE (**A**daptive, **T**rustworthy, **M**anageable, **O**rchestrated, **S**ecure, **P**rivacy-assuring, **H**ybrid, **E**cosystem for **RE**silient Cloud Computing), RNP e MCTIC, no âmbito to acordo de cooperação Número 51119. *ATMOSPHERE is funded by the European Commission under the Cooperation Programme, Horizon 2020 grant agreement No 777154*. Agradecem também o apoio da Fapesp, por meio do projeto temático *Segurança e Confiabilidade da Informação: Teoria e Aplicações*, no. 2013/25.977-7.

Referências

[CRI 2012] (2012). The CRIME attack: netifera.com. *URL: http://netifera.com*.

[BRE 2013] (2013). The BREACH attack: SSL GONE IN 30 SECONDS. *URL: http://breachattack.com*.

[Log 2016] (2016). The logjam attack and weak diffie-hellman. *URL: https://weakdh.org*.

[Adrian et al. 2015] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L., and Others (2015). Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17. ACM.

[AlFardan et al. 2013] AlFardan, N. J., Bernstein, D. J., Paterson, K. G., Poettering, B., and Schuldt, J. C. N. (2013). On the security of rc4 in tls. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 305–320, Berkeley, CA, USA. USENIX Association.

[Alkemade 2013] Alkemade, T. (2013). Mitigating the beast attack on tls. *URL: https://blog.thijsalkema.de/blog/2013/10/08/piercing-through-whatsapp-s-encryption*.

[Apache Software Foundation 2019] Apache Software Foundation (2019). Apache HTTP Server Project. *URL: http://httpd.apache.org*.

[Aviram et al. 2016] Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J. A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., and Shavitt, Y. (2016). DROWN: Breaking TLS using SSLv2. *Proceedings of the 25th USENIX Security Symposium*, (August):1–18.

[Bernstein 2006] Bernstein, D. J. (2006). Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer.

[Bernstein et al. 2013] Bernstein, D. J., Lange, T., et al. (2013). Safecurves: choosing safe curves for elliptic-curve cryptography. *URL: http://safecurves.cr.yp.to*.

[Bleichenbacher 1998] Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer.

[BlueKrypt] BlueKrypt. Cryptographic Key Length Recommendation. *URL: http://www.keylength.com*.

[Braga and Dahab 2015] Braga, A. and Dahab, R. (2015). Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software. In *Caderno de minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, pages 1–50. Sociedade Brasileira de Computação.

[Braga and Dahab 2018] Braga, A. and Dahab, R. (2018). Criptografia assimétrica para programadores - evitando outros maus usos da criptografia em sistemas de software. In *Caderno de minicursos do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2018*, pages 1–50. Sociedade Brasileira de Computação.

[Calvo 2018] Calvo, R. (2018). The true cost of certificate authority trials: Can you trust them? *URL: https://www.isc2.org/News-and-Events/Infosecurity-Professional-Insights*.

[Chandra et al. 2002] Chandra, P., Messier, M., and Viega, J. (2002). Network security with OpenSSL. *O'Reily*.

[Diffie and Hellman 1976] Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654.

[Eldewahi et al. 2015] Eldewahi, A. E. W., Sharfi, T. M. H., Mansor, A. A., Mohamed, N. A. F., and Alwahbani, S. M. H. (2015). Ssl/tls attacks: Analysis and evaluation. In *2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE)*, pages 203–208.

[Fardan and Paterson 2013] Fardan, N. A. and Paterson, K. (2013). Lucky thirteen: Breaking the TLS and DTLS record protocols. *Security and Privacy (SP), 2013 IEEE Symposium on (2013)*.

[Gluck et al. 2013] Gluck, Y., Harris, N., and Angel, A. (2013). BREACH: Reviving the CRIME Attack. In *Black Hat Conference*.

[Hankerson et al. 2004] Hankerson, D., Vanstone, S., and Menezes, A. (2004). *Guide to elliptic curve cryptography*.

[Jonsson and Burt Kaliski 2003] Jonsson, J. and Burt Kaliski (2003). RSA Laboratories Public-Key Cryptography Standards (PKCS)#1: RSA Cryptography Specifications Version 2.1. *URL: https://tools.ietf.org/html/rfc3447*.

[Koblitz 1987] Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209.

[Miller 1985] Miller, V. S. (1985). Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer.

[mod_ssl 2019] mod_ssl (2019). mod_ssl:the apache interface to openssl. *URL: http://www.modssl.org*.

[Möller et al. 2014] Möller, B., Duong, T., and Kotowicz, K. (2014). The POODLE attack. *URL: https://www.openssl.org/ bodo/ssl-poodle.pdf*.

[mozilla 2018] mozilla (2018). Mozilla's server side tls guidelines. *URL: https://wiki.mozilla.org/Security/Server_Side_TLS*.

[mozilla 2019] mozilla (2019). Http and ssl observatory. *URL: https://observatory.mozilla.org*.

[NIST 2001] NIST (2001). Recommendation for block cipher modes of operation. *NIST SP 800-38A URL: http:csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf*.

[NIST 2007] NIST (2007). Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. *NIST SP 800-38D URL: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf*.

[NIST 2012] NIST (2012). Recommendation for Key Management – Part 1: General (Revision 3). *URL: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf*.

[NIST 2013] NIST (2013). Digital Signature Standard (DSS).

[OpenSSL.org] OpenSSL.org. OpenSSL Cryptography and SSL/TLS toolkit. *URL: OpenSSL.org*.

[OWASP 2015] OWASP (2015). OWASP Testing Project v4. *URL: https://www.owasp.org/index.php/OWASP_Testing_Project*.

[Pornin 2017] Pornin, T. (2017). Testsslserver. *URL: https://www.bolet.org/TestSSLServer*.

[Qualys 2019] Qualys, I. (2019). Qualys ssl labs. *URL: https://www.ssllabs.com*.

[rbsec 2019] rbsec (2019). Sslscan. *URL: https://github.com/rbsec/sslscan*.

[Rescorla 2018] Rescorla, E. (2018). The transport layer security (tls) protocol version 1.3. *URL: https://tools.ietf.org/html/rfc8446*.

[Ristic 2011] Ristic, I. (2011). Mitigating the beast attack on tls. *URL: https://blog.qualys.com/ssllabs/2011/10/17/mitigating-the-beast-attack-on-tls*.

[Ristic 2015] Ristic, I. (2015). *OpenSSL cookbook (Version 2.1-draft, June 2018)*. Feisty Duck, 2nd. edition.

[Rivest et al. 1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[SEC 2000] SEC, S. (2000). Sec 2: Recommended elliptic curve domain parameters, version 1. *Standards for Efficient Cryptography Group, Certicom Corp (http://www.secg.org/)*.

[SEC 2010] SEC, S. (2010). Sec 2: Recommended elliptic curve domain parameters, version 2. *Standards for Efficient Cryptography Group, Certicom Corp ()http://www.secg.org/*.

[SSL Labs 2019] SSL Labs (2019). Ssl and tls deployment best practices. *URL: https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices*.

[Stallings 2003] Stallings, W. (2003). *Cryptography and network security, principles and practices*.

[Sullivan 2018] Sullivan, N. (2018). A detailed look at rfc 8446 (a.k.a. tls 1.3). *URL: https://blog.cloudflare.com/rfc-8446-aka-tls-1-3*.

[Synopsys 2014] Synopsys (2014). The Heartbleed Bug. *URL: http://heartbleed.com*.

[Wikipedia 2018] Wikipedia (2018). Netscape. *URL: https://en.wikipedia.org/wiki/Netscape*.