Bundesamt
für Sicherheit in der
Informationstechnik

Eine Studie im Auftrag des
Bundesamtes für Sicherheit in der Informationstechnik

Projekt 154

# Quellcode-basierte Untersuchung von kryptographisch relevanten Aspekten der OpenSSL-Bibliothek

## Arbeitspaket 2:
## Random Number Generator

Version 1.2.1 / 2015-11-03

Sirrix AG
security technologies

## Zusammenfassung

Die Kryptobibliothek OpenSSL wird seit mehreren Jahren in verschiedenen kryptographischen Systemen eingesetzt. Sie ist besonders wichtig bei gesicherten Datenübertragungen in Computernetzwerken und spielt aufgrund ihrer großen Verbreitung im Internet eine besonders wichtige Rolle bei der Verwendung von HTTPS (HTTP over SSL/TLS). OpenSSL bietet eine große Zahl von kryptographischen Funktionen an; dazu gehören z.B. symmetrische und asymmetrische Datenverschlüsselung, digitale Signaturen, Authentisierung, Hashfunktionen, Erstellung und Verifikation von Zertifikaten inklusive Zertifikatsketten, Mechanismen zum Integritätsschutz, Schlüssel- und Zufallszahlenerzeugung. Die wichtigste Aufgabe von OpenSSL ist es jedoch, eine Implementierung des TLS-Protokolls (früher SSL-Protokoll) zu sein. Damit stellt die Bibliothek sowohl grundlegende Kryptofunktionen als auch das High-Level-Protokoll TLS zur Verfügung.
Dieser Bericht ist das Ergebnis der Untersuchung des Zufallszahlengenerators (RNG) der OpenSSL-Bibliothek. Es wird die generelle Einbindung des RNGs in die API, die Verwendung sowie Nicht-Verwendung in anderen Bibliotheks-Funktionen, sowie die Implementierung des Standard-RNGs beschrieben.

**Autoren**
Wolfgang Meyer zu Bergsten (MzB), Sirrix AG
René Korthaus (RK), Sirrix AG

# Quellcode-basierte Untersuchung von kryptographisch relevanten Aspekten der OpenSSL-Bibliothek

OpenSSL 1.0.1g

# Inhaltsverzeichnis

# 1 OpenSSL Zufallszahlengenerator (RNG)

Der Zufallszahlengenerator (RNG) einer Kryptobibliothek stellt einen unverzichtbaren, äußerst sicherheitskritischen Anker dar. Zufallszahlen werden in OpenSSL u.a. für die Erzeugung von Schlüsseln, Initialisierungsvektoren und für das Blinding bei asymmetrischen Ver- und Entschlüsselungen genutzt. Daher legen wir einen Schwerpunkt unserer Analyse auf den RNG, dessen Seeding- und Verwaltungsfunktionen. Darüber hinaus wird die Qualität der Implementierung und der Schnittstellen untersucht.

Der OpenSSL RNG ist ein Pseudozufallszahlengenerator, der auf komplexitätstheoretischen Annahme basiert, dass die verwendete Mischfunktion SHA-1 nicht-invertierbar ist. In OpenSSL werden Daten, die nach ausreichendem Seeden (d.h. mit 32 Bytes Entropie, s. Abs. 2.3) des RNG-Zustands gemäß OpenSSL-Richtlinie entnommen werden, als „kryptografisch starke“ („cryptographically strong“) Zufallsdaten bezeichnet. „Kryptografisch schwach“ bedeutet im Gegensatz dazu, dass der RNG-Zustand zum Zeitpunkt der Entnahme der Zufallsdaten noch nicht ausreichend geseedet wurde.

## 1.1 Interface für den RNG

OpenSSL ermöglicht über definierte Interfaces die anwenderspezifische Implementierung von RNGs. Das von RNGs zu implementierende Interface stellt Funktionen für das Hinzufügen von Entropie, die Entnahme sowie die Abfrage des RNG-Status in der Struktur RAND_METHOD bereit.

```
struct rand_meth_st
     {
     void (*seed)(const void *buf, int num);
     int (*bytes)(unsigned char *buf, int num);
     void (*cleanup)(void);
     void (*add)(const void *buf, int num, double entropy);
     int (*pseudorand)(unsigned char *buf, int num);
     int (*status)(void);
     };
typedef struct rand_meth_st RAND_METHOD;
```

| API Funktion | RAND_METHOD Element | Kurzbeschreibung |
|---|---|---|
| RAND_seed(buf, num) | seed(buf, num) | Seeden des RNG-Zustands |
| RAND_add(buf, num, entropy) | add(buf, num, entropy) | Seeden des RNG-Zustands mit Angabe der geschätzten Entropie |
| RAND_status() | status() | Feststellen, ob der RNG genutzt werden kann |
| RAND_bytes(buf, num) | bytes(buf, num) | Lesen von kryptografisch starken Zufallsdaten |
| RAND_pseudo_bytes(buf, num) | pseudorand(buf, num) | Lesen von evtl. kryptografisch schwachen Zufallsdaten |
| RAND_cleanup() | cleanup() | Zurücksetzen des RNGs |

*Tabelle 1.1: OpenSSL-API-Funktionen für den RNG und deren Abbildung auf den Standard-RNG*

Die Abbildung der Zufallszahlenfunktionen der API auf eine konkrete Implementierung erfolgt über die Funktion RAND_get_rand_method() (s. Abschnitt 2.1).

### 1.1.1 Rückgabewerte der RNG-Funktionen

Die korrekte Nutzung der Funktionen des RNG setzt voraus, dass die Rückgabewerte korrekt interpretiert werden. Die Rückgabewerte der API-Funktionen wurden gemäß der OpenSSL-Dokumentation in der folgenden Tabelle herausgearbeitet.

| Funktion | Rückgabewert bei Erfolg | Rückgabewert bei Fehler |
|---|---|---|
| RAND_seed() | kein | kein |
| RAND_status() | kein | kein |
| RAND_cleanup() | kein | kein |
| RAND_status() | 1 (RNG mit 32 Bytes Entropie geseedet, s. Abs. 2.3) | 0 |
| RAND_bytes() | 1 | 0: keine RNG-Daten erzeugt<br>-1: interner Fehler |
| RAND_pseudo_bytes() | 1: Daten kryptografisch stark<br>0: Daten nicht kryptografisch stark | -1: interner Fehler |

*Tabelle 1.2: Rückgabewerte der RNG-API*

Die Extraktion der Daten erfolgt für RAND_bytes() und RAND_pseudo_bytes() auf gleiche Weise. Somit verhalten sich die Funktionen bei den Rückgabewerten „-1“ und „1“ identisch. Lediglich die Bedeutung des Rückgabewerts „0“ ist unterschiedlich.

Für eine korrekte Verwendung der API mussbei RAND_bytes() und RAND_status() auf !=1 getestet

werden, bei RAND_pseudo_bytes hingegen auf < 0.

**Bemerkung 1:** Der Rückgabewert der RAND_bytes() und RAND_pseudo_bytes()-Funktion ist nicht konsistent. Gerade die Unterschiede in der Fehlerbehandlung können zu signifikanten Fehlern führen: Der Rückgabewert „0“ zeigt bei RAND_pseudo_bytes() gewünschtes, bei RAND_bytes() unerwünschtes Verhalten an. Wenn eine Funktion durch die andere ersetzt wird, so muss auch die Fehlerbehandlung angepasst werden. Dieses Verhalten ist nicht intuitiv, da die Verfügbarkeit der unterschiedlichen Funktionen einheitliche Behandelbarkeit suggeriert.

**Empfehlung 1:** Der Rückgabewert von RAND_pseudo_bytes() sollte bei pseudozufälligen Daten ebenfalls den Rückgabewert „1“ liefern. Damit verschwindet zwar die Unterscheidbarkeit von kryptografisch starken und schwachen Daten für diese Funktion, diese kann in Programmen aber durch Aufruf von RAND_bytes() dennoch umgesetzt werden. Hierdurch wird die Intention des Programmierers explizit gemacht und der Programmcode wird besser wartbar.

### 1.1.2 Spezifizieren einer RNG-Implementierung

Die Einbindung von einer RNG-Implementierung kann auf zwei Wegen erfolgen

1. Empfohlen: RAND_set_rand_engine(): Diese Variante nutzt das Engine-Interface, um die RNG-Methode zu spezifizieren.
2. Überholt: RAND_set_rand_method(): Diese Variante setzt die RNG-Methode direkt. Eine so gesetzte RAND_METHOD wird allerdings ignoriert, sobald eine ENGINE-Implementierung spezifiziert wird.

**Bemerkung 2:** Sofern eine RAND_METHOD durch RAND_set_rand_method() gesetzt wird, muss überprüft werden, ob diese nicht durch eine ENGINE-Implementierung ersetzt wird. Dies kann zu Verhalten führen, das vom Programmierer nicht gewünscht ist.

**Empfehlung 2:** Es sollte ausschließlich das RAND_set_rand_engine-Interface mit einer ENGINE-Implementierung für die Spezifizierung des RNG genutzt werden. Alter Code sollte auf das ENGINE-Interface portiert werden.

### 1.1.3 Speichern von Entropie zwischen zwei Programmläufen

Zum Speichern von Entropie zwischen zwei Programmläufen bietet OpenSSL die Funktionen RAND_write_file() und RAND_load_file(), sowie die Funktion RAND_file_name(). Durch die Verwendung dieser Funktionen kann das Seeding beim Programmstart reduziert oder sogar darauf verzichtet werden. Dies ist insbesondere sinnvoll, wenn das OpenSSL nutzende Programm auf Plattformen eingesetzt wird, die wenig Entropie liefern, oder zu einem Zeitpunkt gestartet wird, zu dem noch wenig Entropie verfügbar ist. Ein typisches Beispiel für einen solchen Fall ist ein Webserver, der sofort nach dem Systemstart gestartet wird.

| Funktion | Zweck |
|---|---|
| int RAND_load_file(const char *file, long bytes) | Lesen von Daten mit Entropie aus Datei |
| int RAND_write_file(const char *file) | Schreiben von Daten aus dem RNG-Zustand in Datei |

Das Speichern der Entropie erfolgt über die Funktion RAND_write_file(). Mit dieser Funktion werden 1024 Bytes mit Hilfe der RAND_bytes()-Funktion aus dem RNG-Zustand gelesen und in einer Datei gespeichert. Es werden keine Informationen über die im Pool verfügbare Entropie gespeichert. Im Normalfall gibt RAND_write_file() die Anzahl der geschriebenen Bytes zurück. Falls nicht ausreichend Entropie im RNG-Zustand enthalten ist, werden keine Daten geschrieben und „-1“ zurückgeben.

Das Lesen der Entropie erfolgt über die Funktion RAND_load_file(). Hierbei wird eine bestimmte Anzahl Bytes aus der spezifizierten Datei gelesen und in den RNG-Zustand mit der RAND_add()-Funktion eingespeist. Die Entropie der Datei wird mit 100% angenommen. Beim Aufruf der Funktion kann angegeben werden, wie viele Bytes maximal gelesen werden sollen. Der RNG-Zustand wird mit den tatsächlich gelesenen Bytes geseedet, wobei die Entropieschätzung 100% beträgt. Als Rückgabewert wird die Anzahl von Bytes geliefert, die aus der Datei gelesen wurden.

| Datei | Funktion | Bemerkung |
|---|---|---|
| apps/app_rand.c | **RAND_write_file()**<br>**RAND_load_file()** | Dienstfunktion für Kommandozeilenwerkzeug „openssl“. Wird dort bei der Initialisierung des RNG verwendet. Nicht detailiert untersucht. |
| apps/winrand.c | **RAND_write_file()**<br>**RAND_load_file()** | Nur unter Windows genutzt, hier nicht untersucht |
| demos/easy_tls/easy-tls.c | **RAND_load_file()** | Beispielanwendung für TLS. Nicht detailliert untersucht. |

*Tabelle 1.3: Verwendung der RAND_write_file() und RAND_load_file() Funktionen*

Das Lesen und Speichern von Daten aus/in eine Datei muss explizit vom OpenSSL-nutzenden Programm aufgerufen werden und erfolgt nicht implizit. Die Verwendungen gemäß Tabelle Tabelle 1.3 sind nicht im Bibliotheksteil von OpenSSL, sondern im mitgelieferten Kommandozeilenwerkzeug sowie einer Demo-Implementierung.

**Bemerkung 3:** Beim Einspeisen der Entropie in den RNG-Zustand wird aufgrund der Implementierung eine Entropie von 100% angenommen. Dies muss aber nicht der tatsächlich enthaltenen Entropie entsprechen, da bei einer Einspeisung von 32 Bytes Entropie in den RNG-Zustand automatisch von RAND_write_file() 1024 Bytes in die Datei geschrieben werden, und es keine Möglichkeit gibt, nur die tatsächlich vorhandene Menge Entropie in die Datei zu schreiben. Beispiel: Seeding des RNG-Zustands mit 32 Bytes, dann Speichern mittels RAND_write_file() in Datei. Wenn dann

beim nächsten Start mehr als 32 Bytes aus der Datei gelesen werden, stimmt die Entropieschätzung aufgrund der Entropieannahme von 100% in der Datei nicht mehr.

**Empfehlung 3:** Die tatsächlich im RNG-Zustand vorhandene Schätzung für die in den RNG-Zustand eingebrachte Entropie muss beim Export gespeichert und beim Import wiederhergestellt werden.

**Bemerkung 4:** Die Entropie wird persistent gespeichert, und nach dem Lesen nicht gelöscht. Als Folge könnte ein Programm, das bei Beendigung diese Datei nicht neu schreibt, den RNG-Zustand mehrmals mit den gleichen Daten seeden. Beispielsweise kann dieser Fall auftreten, wenn das Programm irregulär beendet wird. Dies ist insbesondere problematisch, wenn Entropie aus der Datei die einzige Entropiequelle ist. Da die OpenSSL-Dokumentation das Speichern mit diesen Funktionen explizit als Ersatz für Seeding mit frischen Daten empfiehlt („Its state can be saved in a seed file (see RAND_load_file(3)) to avoid having to go through the seeding process whenever the application is started.“), ist diese Vorgehensweise sogar wahrscheinlich.

**Empfehlung 4:** Die Datei zur Speicherung der Entropie sollte direkt nach dem Lesen der Daten gelöscht werden, um eine Wiederverwendung des Seeds zu verhindern.

**Bemerkung 5:** Analog zur vorigen Darstellung kann gleichzeitiger Zugriff verschiedener Programme auf die Datei zu Seeding mit identischen Daten führen, wenn die gleiche Datei (insbes. die von RAND_file_name() referenzierte) verwendet wird.

**Empfehlung 5:** Jedes Programm sollte eine eigene Entropie-Datei spezifizieren und im Programmablauf sicherstellen, dass die Datei nur einmal gelesen wird. Zusätzlich sollte eine weitere Entropiequelle zum Seeden des RNG genutzt werden.

**Sicherheitskritische Daten**

- Die Puffer zum Lesen und Schreiben der Entropie aus der/in die Datei. Diese werden sicher mit OPENSSL_cleanse() (s. Abschnitt 2.5) überschrieben.

**Diskussion**: Unter der Voraussetzung, dass Daten aus /dev/random und /dev/urandom hohe Entropie aufweisen, schwächen die angesprochenen Probleme auf einem typischen System den RNG nicht derart, dass daraus entnommene Zufallsdaten schwach sind. Der Grund dafür ist inbesondere das implizite Seeden des RNG mit weiteren Quellen (insbes. /dev/random und /dev/urandom). Die Schätzung für die in den RNG-Zustand eingebrachte Entropie kann beim Speichern der Entropie zwischen Programmläufen mit den dafür vorgesehenen Funktionen zu hoch sein.

Im Worst-Case Szenario allerdings, wenn:

- /dev/random nicht verfügbar ist,
- /dev/urandom Daten mit niedrigem Entropiegehalt liefert,

- und die Entropiedatei eine falsche Entropieschätzung enthält,

wird der OpenSSL-RNG nicht ausreichend geseedet und kann daher keine nicht-vorhersagbaren Zufallszahlen erzeugen.

### 1.1.4 Standard-RNG der OpenSSL-Bibliothek

Der Standard-RNG der OpenSSL Bibliothek ist RAND_SSLeay. Dieser wird nicht über das engine-Interface eingebunden, sondern ist der Standard-Fallback in der Funktion RAND_get_rand_method() (s. Abschnitt 2.1).

## 1.2 Testsuite für den RNG

Um die Qualität des Zufallszahlengenerators zu bewerten sind statistische Tests notwendig. OpenSSL liefert in seiner Testsuite einen einfachen Test mit. Desweiteren existiert nur ein Laufzeit-Test, in dem geprüft wird, ob der RNG mit ausreichend Entropie geseedet wurde.

### 1.2.1 Testsuite

Der Test für den RNG ist in test/randtest.c implementiert. Es wird lediglich die Funktion RAND_pseudo_bytes() getestet. Von dieser Funktion wird getestet:

- ob ein RNG verfügbar ist
- Power-Up-Tests nach FIPS 140-1:
  - Monobit Test,
  - Poker Test,
  - Runs Test,
  - Long Run Test.

RAND_bytes() und RAND_pseudo_bytes() werden vom identischen Codepfad implementiert. Der einzige Unterschied der beiden Funktionen ist der Rückgabewert in dem Fall, dass entropy < ENTROPY_NEEDED ist.

**Bemerkung 6:** Diese Tests stellen statistische Grundeigenschaften des RNG fest. Sie reichen aber nicht aus, um die Eignung des RNG für kryptografische Zwecke sicherzustellen. Dieses ist ein grundsätzliches Problem bei Zufallstests. Die Existenz dieser Tests mit der Erfolgsmeldung könnte den Nutzer in falscher Sicherheit wiegen.

**Empfehlung 6:** Der Test sollte auf den Umstand hinweisen, dass ein erfolgreicher Test des RNG lediglich statistische Grundeigenschaften, nicht aber kryptografische Sicherheit feststellt.

**Bemerkung 7:** Da kein Test für den kryptografisch sicheren Zufallszahlengenerator RAND_bytes()

existiert, wird nicht sichergestellt, dass dessen Fail-Safe-Funktionalität wie erwartet arbeitet. So könnte beispielsweise Schlüsselmaterial erzeugt werden, obwohl der RNG-Zustand nicht ausreichend geseedet wurde. Daher ist die Verifikation dieser Funktionalität von großer Bedeutung.

**Empfehlung 7:** Implementierung eines deterministischen Tests, bei dem die Menge der Entropie vorgegeben wird und daraus die damit zu erzeugende Menge an Zufallsdaten berechnet werden kann. Es muss überprüft werden, ob RAND_bytes() einen Fehlerwert zurück gibt, wenn der RNG noch nicht ausreichend geseedet wurde. Das implizite Seeding von OpenSSL muss hierfür deaktiviert werden.

### 1.2.2 Laufzeit-Tests

Während der Laufzeit wird lediglich getestet, ob der RNG mit ausreichend Entropie geseedet wurde. Sofern genug Entropie vorhanden ist, können kryptografisch starke Zufallszahlen entnommen werden, ansonsten kann nur die Funktion für pseudozufällige Daten aufgerufen werden.

**Bemerkung 8:** Es existieren keine Tests, mit denen die Integrität des RNGs zur Laufzeit getestet werden kann.

**Empfehlung 8:** Es sollte ein Test hinzugefügt werden, mit dem vor der Initialisierung des RNGs anhand von bekannten Seeds und Testdaten die Integrität des RNGs festgestellt werden kann.

## 1.3 High-Level Analyse des Standard-RNG

Für die Analyse wurde zunächst Black-Box-Tests durchgeführt. Für diese Tests wurde die offizielle API-Dokumentation von OpenSSL als Referenz herangezogen. Darauf basierend wurden verschiedene Tests implementiert, mit denen wichtige Eigenschaften eines RNGs analysiert wurden.

Im zweiten Schritt wurden verschiedene Werkzeuge für die statische Analyse auf den Code angewendet. Eine solche Analyse findet schnell Fehler bzw. Warnungen, die dann analysiert werden können.

Abschließend wurde ein Source Code Review durchgeführt. Dadurch wurde Verständnis der internen Struktur erlangt sowie Schwächen in der Implementierung festgestellt. Darüber hinaus konnte mit dem so erlangten Wissen ein Modell für den RNG erstellt werden.

### 1.3.1 BlackBox-Tests

Die Analyse des Zufallszahlengenerators wurde mit einem Blackbox-Test begonnen. Dafür wurden auf dem Interface basierende Tests geschrieben.

#### *1.3.1.1 Der RNG kann ohne explizites Seeden genutzt werden*

Aus der Dokumentation geht hervor, dass der RNG bei Verfügbarkeit von /dev/urandom automatisch Seed aus /dev/urandom liest. Wenn /dev/urandom nicht verfügbar ist, liefert der RNG dennoch

Zufallsdaten. Weitere Untersuchung zeigt, dass auch /dev/random genutzt wird, um den RNG zu seeden (s. Tabelle Tabelle 1.4). Dies steht im Gegensatz zur OpenSSL-Dokumentation, in der nur /dev/urandom erwähnt wird (s. https://www.openssl.org/docs/crypto/RAND_add.html).

Wenn weder /dev/random noch /dev/urandom lesbar sind, verhält sich der RNG wie in der Dokumentation beschrieben und gibt bei der Entnahme von Zufallsdaten einen Fehlerwert zurück.

**Bemerkung 9:** Gemäß Dokumentation soll /dev/urandom nicht als kryptografisch starke Zufallsquelle angesehen werden. Dennoch ist /dev/urandom als einzige Entropiequelle ausreichend, um mit RAND_bytes() vermeintlich kryptografisch sichere Zufallsdaten zu liefern.

**Empfehlung 9:** /dev/urandom sollte als Entropiequelle deaktiviert werden.

Das Lesen aus /dev/random und /dev/urandom erfolgt in einer Schleife, in der bis zu 10 ms lang versucht wird, die notwendige Menge an Zufallsdaten zu lesen. Der RNG-Zustand wird anschließend mittels RAND_add() geseedet, wobei die Entropieschätzung 100% beträgt. Wenn weniger als die geforderte Menge an Daten gelesen werden kann, werden die tatsächlich gelesenen Daten mit einer Entropieschätzung von 100% in den RNG-Status eingemischt. Der Entropiezähler wird ebenfalls nur um die tatsächlich gelesene Entropie erhöht. Daher ist der Wert des Entropieschätzers weiterhin korrekt.

| **verfügbare Kernel-Quelle** | | **OpenSSL-Funktionen** | | |
|---|---|---|---|---|
| /dev/urandom | /dev/random | RAND_bytes | RAND_pseudo_bytes | RAND_status |
| ja | ja | erfolgreich (1) | erfolgreich (1) | erfolgreich (1) |
| ja | nein | erfolgreich (1) | erfolgreich (1) | erfolgreich (1) |
| nein | ja | erfolgreich (1) | erfolgreich (1) | erfolgreich (1) |
| nein | nein | Fehler (-1) | erfolgreich (0) | Fehler (0) |

*Tabelle 1.4: Abhängigkeit der Entropieentnahme von verfügbaren Entropiequellen. Der Wert in Klammern beschreibt den Rückgabewert.*

**Bemerkung 10:** Selbst wenn keinerlei Entropie in den RNG-Zustand eingebracht wurde, liefert die Funktion RAND_pseudo_bytes() Daten. Diese werden dann, je nach Zustand des RNGs, entweder aus den Daten erzeugt, die beim Programmstart zufällig im Speicher des RNGs standen, oder aus Nullen, die nach einem Reset des RNG in den RNG-Zustand geschrieben werden.

**Empfehlung 10:** Die Verwendung von RAND_pseudo_bytes() ist nicht sicher. Wir empfehlen, auf die Verwendung von RAND_pseudo_bytes() zu verzichten.

#### 1.3.1.2 Seeding und Entnahme von kryptografisch starker Entropie

Als nächstes wurde getestet, mit wie viel Bytes Entropie der RNG geseedet werden muss, bis der Rückgabewert von RAND_bytes() gleich 1 ist. Dabei zeigte sich, dass nach dem Seeden mit 32 By-

tes Entropie unbegrenzt mittels RAND_bytes() und RAND_pseudo_bytes() Zufallsdaten entnommen werden können (Getestet wurde die Entnahme von mehr als 10 TeraByte).

Die Menge für die Entnahme von Daten pro Aufruf wird generell durch das Interface der Funktion auf $2^{31}$-1 Bytes beschränkt, da der Parameter für die Menge zu entnehmender Daten vom Typ „int“ ist (Wertebereich: $-2^{31}...2^{31}$-1).

Es ist irrelevant, ob die Entropie über die RAND_seed() oder RAND_add() Funktion eingespeist wird.

#### 1.3.1.3 Weitere in den RNG-Zustand eingepflegte Daten

Daten, die aus dem RNG-Zustand entnommen werden, unterscheiden sich auch bei identischem Seed. Daher liegt die Vermutung nahe, dass implizit weitere Quellen in den RNG-Zustand eingepflegt werden.

Die Ursache für die Unterschiede kann hier nicht abschließend festgestellt werden. Um diese Analyse weiterzuführen, ist ein Source Code Review notwendig. Eine vollständige Liste von Entropiequellen wurde durch das Source Code Review festgestellt und ist in Tabelle Tabelle 2.1 dargestellt.

### 1.3.2 Statische Analyse

Für die statische Analyse wurden die Werkzeuge cppcheck, clang-analyzer und flexelint eingesetzt.

#### 1.3.2.1 cppcheck

**Konfiguration**

cppcheck -j 8 -i doc -i MacOS/ -i Netware/ -i VMS --force --enable=all

**Ergebnisse**

cppcheck hat im Rand-Subsystem keine Fehler gefunden.

#### 1.3.2.2 clang-analyzer

**Konfiguration**

Für die Analyse wurden mit Ausnahmen der Checker für OS X alle funktionierenden Checker von clang-analyzer, Version 3.4, aktiviert. Diese sind:

core.AdjustedReturnValue
core.AttributeNonNull
core.builtin.BuiltinFunctions
core.builtin.NoReturnFunctions
core.CallAndMessage
core.DivideZero
core.DynamicTypePropagation
core.NonNullParamChecker
core.NullDereference
core.StackAddressEscape
core.UndefinedBinaryOperatorResult
core.uninitialized.ArraySubscript
core.uninitialized.Assign
core.uninitialized.Branch
core.uninitialized.CapturedBlockVariable
core.uninitialized.UndefReturn
core.VLASize
cplusplus.NewDelete
deadcode.DeadStores
llvm.Conventions
security.FloatLoopCounter
security.insecureAPI.getpw
security.insecureAPI.gets
security.insecureAPI.mkstemp
security.insecureAPI.mktemp
security.insecureAPI.rand
security.insecureAPI.strcpy
security.insecureAPI.UncheckedReturn
security.insecureAPI.vfork
unix.API
unix.cstring.BadSizeArg
unix.cstring.NullArg
unix.Malloc
unix.MallocSizeof
unix.MismatchedDeallocator

#### Ergebnisse

clang-analyzer hat im Rand-Subsystem keine Fehler gefunden.

### *1.3.2.3 FlexeLint*

#### Konfiguration

-w2 (Error und Warning aktiviert)

#### Ergebnisse

FlexeLint bemängelt das Ignorieren von Rückgabewerten.

- md_rand.c: beim Aufruf verschiedener Hash-Funktionen
- md_rand.c: beim Aufruf von RAND_poll()
- randfile.c: beim Aufruf von setvbuf(), chmod(), BUF_strlcpy(), BUF_strlcat()

Die Untersuchung dieser Warnung und der Analyse, ob sie eine Schwäche darstellen, erfolgt im folgenden Review-Teil (Abs. 2.2).

# 2 Source Code Review

Für die detaillierte Bewertung des OpenSSL RNG wurde dessen Implementierung einem Source Code Review unterzogen.

Die Implementierung ist auf verschiedene Dateien aufgeteilt. So findet sich der algorithmische Kern des Standard-RNGs in der Datei crypto/rand/md_rand.c. Linux-spezifische Entropiequellen sind in crypto/rand/rand_unix.c implementiert. Den Adapter für das OpenSSL-API-Interface bildet die Datei crypto/rand/rand_lib.c.

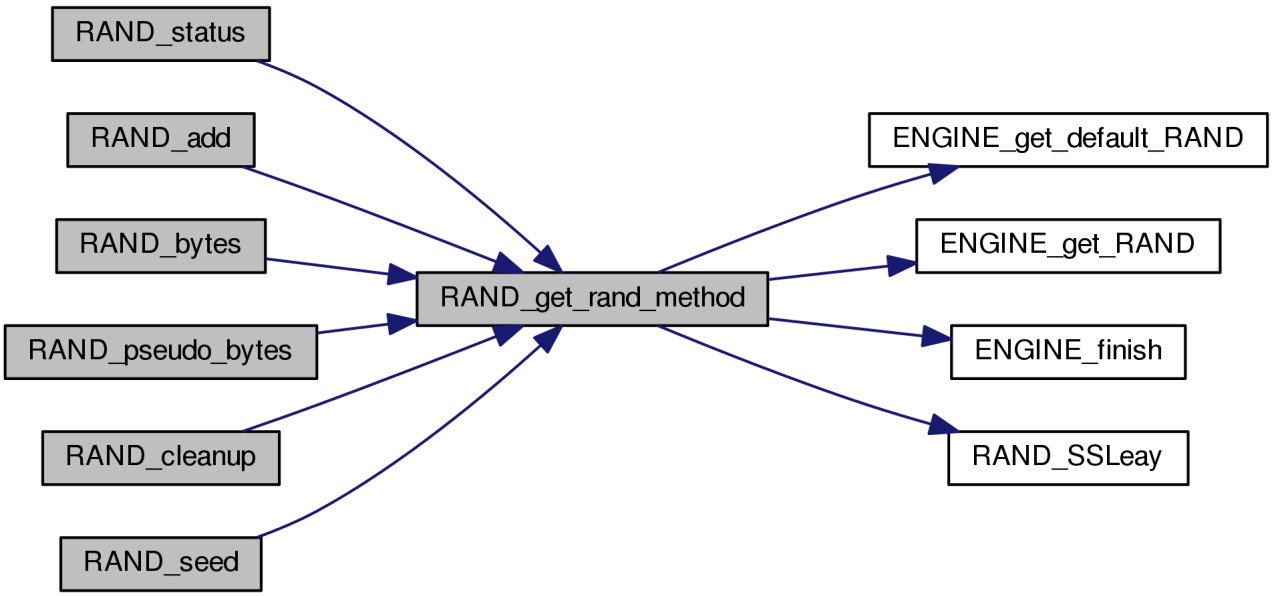
Außerhalb dieser Dateien ist die Funktion OPENSSL_cleanse() relevant, die sicheres Überschreiben von Daten sicherstellen soll.

| Funktion | Quelldatei |
|---|---|
| OPENSSL_cleanse() | crypto/mem_clr.c |
| ssleay_rand_add() | crypto/rand/md_rand.c |
| ssleay_rand_seed() | crypto/rand/md_rand.c |
| ssleay_rand_bytes | crypto/rand/md_rand.c |
| ssleay_rand_pseudo_bytes() | crypto/rand/md_rand.c |
| ssleay_rand_nopseudo_bytes() | crypto/rand/md_rand.c |
| ssl_rand_cleanup() | crypto/rand/md_rand.c |
| ssleay_rand_status() | crypto/rand/md_rand.c |
| RAND_poll() | crypto/rand/md_unix.c |

Ein Callgraph der Funktionen findet sich in Abschnitt 4.

## 2.1 OpenSSL-Adapter

Die Konvertierung des RAND_SSLeay-Interfaces zu dem OpenSSL-Interface erfolgt über die Implementierung der Funktionen in crypto/rand/rand_lib.c. Der Kern des Adapters ist die Funktion RAND_get_rand_method(), die die RAND_SSLeay-Funktionen zurück gibt, sofern kein Zufallszahlengenerator explizit spezifiziert wurde.




Im Adapter wurden keine Auffälligkeiten gefunden.

## 2.2 Warnungen aus der statischen Analyse

***md_rand.c***

**Bemerkung 11:** Das Ignorieren der Rückgabewerte der Hash-Funktionen kann, je nach Implementierung der Funktion, einen schwerwiegenden Fehler darstellen. Wenn in der Initialisierungsfunktion beispielsweise eine Speicherallokation fehlschlägt, darf danach nicht mit dem fehlerhaft erzeugten Kontext gearbeitet werden.

**Empfehlung 11:** Überprüfung der Rückgabewerte der Hashfunktion einführen.

Das Ignorieren des Rückgabewerts von RAND_poll() stellt kein Problem dar, da der Entropiezähler nur angepasst wird, wenn der Seedingvorgang erfolgreich durchgeführt werden kann.

***randfile.c***

Das Ignorieren des Rückgabewerts von setvbuf(), BUF_strlcpy() und BUF_strlcat() hat keine sicherheitsrelevanten Konsequenzen.

Wenn der Rückgabewert von chmod() ignoriert wird, kann es passieren, dass die Datei für das Speichern von Entropie zwischen zwei Ausführungen von OpenSSL von anderen Benutzern gelesen werden kann.

**Bemerkung 12:** Wenn andere Benutzer die Entropiedatei lesen können, wird hierdurch das Raten des Zustands des RNG-Zustands und damit der erzeugten Zufallsdaten vereinfacht.

**Empfehlung 12:** Überprüfung des Rückgabewerts der chmod()-Operation und Abbruch der Speicherung, wenn die Rechte nicht erfolgreich gesetzt werden können.

## 2.3 Parameter und Zustand des Standard-RNGs

Der Standard-RNG nutzt standardmäßig die Hashfunktion SHA-1 mit einer Digest-Größe von 20 Bytes (s. rand_lcl.h).

Die notwendige Entropie, um hochwertige Zufallsdaten zu liefern, wird in der Datei rand_lcl.h als ENTROPY_NEEDED zu 32 Bytes definiert.

Der Zustand des Standard-RNGs wird in statischen Variablen gespeichert. Damit kann aus allen Funktion der Übersetzungseinheit md_rand.c auf sie zugegriffen werden.

```
#define STATE_SIZE  1023
static int state_num=0,state_index=0;
static unsigned char state[STATE_SIZE+MD_DIGEST_LENGTH];
static unsigned char md[MD_DIGEST_LENGTH];
static long md_count[2]={0,0};
static double entropy=0;
static int initialized=0;
```

| Variable | Verwendung |
|---|---|
| state_num | Anzahl Bytes des RNG-Zustands, die aktuell verwendet werden, 0…1023 |
| state_index | Aktuelle Lese- und Schreibposition im RNG-Zustand, 0…state_num |
| state | RNG-Zustand |
| md | Übertrag von Hashwerten |

| Variable | Verwendung |
|---|---|
| md_count | Lese- und Schreibzähler |
| entropy | Schätzwert für eingebrachte Entropie |
| initialized | Status, ob RNG initialisiert wurde |

**Bemerkung 13:** state ist 1023+MD_DIGEST_LENGTH Bytes groß. Es werden allerdings nur 1023 Bytes genutzt. Die MD_DIGEST_LENGTH zusätzlichen Bytes von state werden weder gelesen noch geschrieben. Dies stellt kein Sicherheitsproblem dar, kann aber darauf hinweisen, dass der Code nicht mehr von einem Programmierer gepflegt wird, der alle Details des RNG versteht.

**Empfehlung 13:** Die Größe des RNG-Zustands sollte auf die notwendige Größe von STATE_SIZE Bytes reduziert werden.

## 2.4 Nebenläufige Programme

Ein zweiter Block statischer Variablen regelt den Zugriff für nebenläufige Programme:

```
static unsigned int crypto_lock_rand = 0;
static CRYPTO_THREADID locking_threadid;
```

| Variable | Verwendung |
|---|---|
| crypto_lock_rand | Flag, ob ein Thread bereits das Lock genommen hat |
| locking_threadid | Thread-ID des Threads mit Lock |

**Bemerkung 14:** Der RNG ist in der Default-Konfiguration nicht Thread-Safe. Um Thread-Safety zu erreichen, müssen die Funktionen locking_function() und threadid_func() mit den Funktionen CRYPTO_set_locking_callback() und CRYPTO_set_id_callback() gesetzt werden.

**Empfehlung 14:** Sofern Threading genutzt wird und aus verschiedenen Threads auf den RNG-Zustand zugegriffen werden soll, müssen die entsprechenden Locking-Funktionen gesetzt werden.

**Bemerkung 15:** Der RNG liefert schwache Entropie, wenn er nach einem fork()-Aufruf nicht erneut geseedet wird. Dies liegt daran, dass die einzige Prozess-spezifische Information die PID des Prozesses ist, die einen beschränkten Bereich umfasst und sich wiederholt (bei einem Standard-Linux-System gibt es PIDs von 0 bis 32768). (Siehe auch http://wiki.openssl.org/index.php/Random_fork-safety)

**Empfehlung 15:** Der RNG muss nach einem Fork neu geseedet werden. Um sicherzustellen, dass

tatsächlich neue Informationen eingepflegt werden, muss der Entropiezähler entropy zurückgesetzt werden. Hierfür stehen die Funktionen ssleay_rand_cleanup() oder RAND_cleanup() zur Verfügung.

**Bemerkung 16:** Der RNG-Zustand state wird als Ringpuffer verwaltet. Wenn Threads gleichzeitig darauf zugreifen, werden ihnen verschiedene Bereiche des RNG-Zustands zum Lesen (Extraktion) und Schreiben (Seeding) der Daten zugewiesen. Aufgrund der beschränkten Größe des Pools (1023 Bytes) überlappen die Datenbereiche ab einer gewissen Anzahl Threads. Durch den gleichzeitigen Zugriff auf den RNG-Zustand ist dessen Zustand nicht mehr deterministisch, und es könnte passieren, dass mehrfach die gleichen Zufallsdaten ausgegeben werden. Dies wird mit einem Geschwindigkeitsvorteil begründet.

**Empfehlung 16:** Das Lock für den RNG-Zustand sollte für die komplette Zeit, in der auf dem RNG-Zustand gearbeitet wird, gehalten werden, und damit der Zugriff auf den RNG-Zustand von mehreren Threads serialisiert werden. Sollte der RNG dadurch für gewisse Anwendungen zu langsam werden, sollten andere Maßnahmen getroffen werden um die Geschwindigkeit zu erhöhen, wie z.B. mehrere Instanzen des RNGs mit unterschiedlichen Seeds zu erzeugen.

## 2.5 OPENSSL_cleanse()

Die Funktion OPENSSL_cleanse() aus crypto/mem_clr.c stellt sicheres Überschreiben von RAM bereit. Die Funktion wird im RNG genutzt, um den RNG-Zustand beim Zurücksetzen sicher zu überschreiben.

Die Funktion ist in einer separaten Übersetzungseinheit implementiert und nutze eine statische Variable. Durch diese beiden Maßnahmen wird sichergestellt, dass ein Compiler den Code nicht durch Optimierungen entfernen kann.

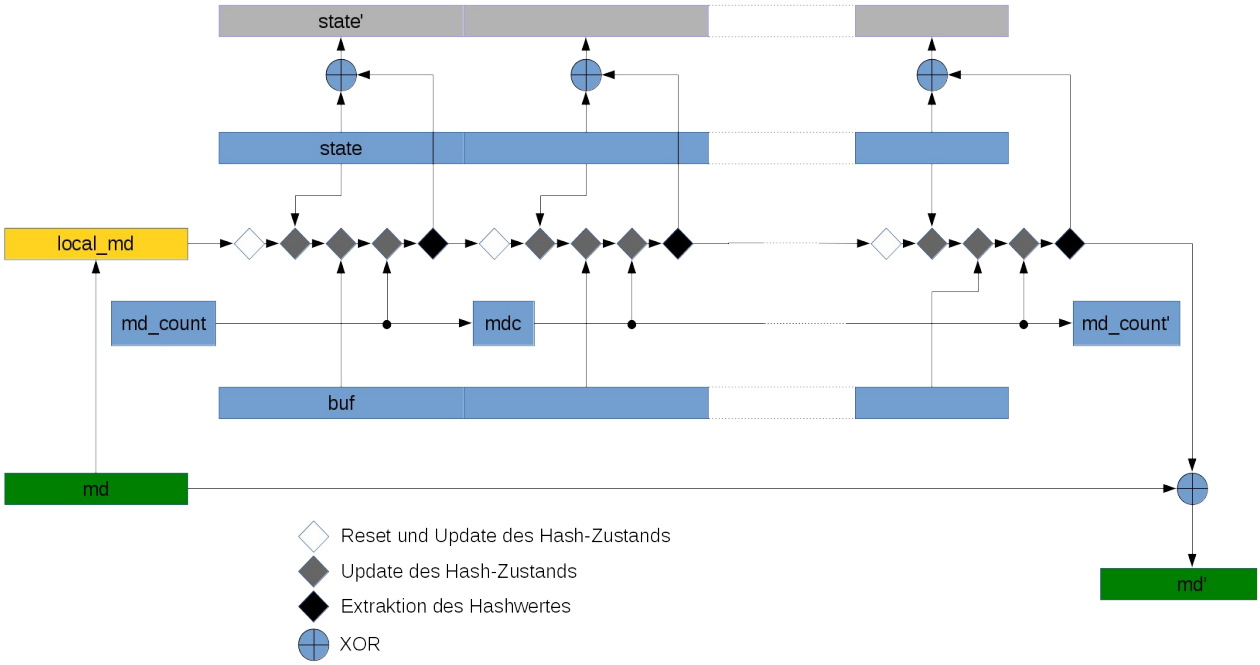
Es wurde mit einem Disassembler festgestellt, dass die OPENSSL_cleanse()-Funktion im erzeugten Code enthalten und wirksam ist. Sie befindet sich in der Bibliothek libcrypto.so/.a.

## 2.6 Analyse der RAND_SSLeay-Funktionen

Der OpenSSL Standard-RNG RAND_SSLeay arbeitet sowohl für die mit RAND_bytes() als auch die mit RAND_pseudo_bytes() entnommenen Daten mit dem identischen Entropiepool. Unterschiede ergeben sich nur in der Erzeugung des Rückgabewerts bei der Entnahme von Zufallsdaten (s. Abs. 2.6.4). Damit wird von diesen Funktionen der selbe RNG-Zustand, der selbe Entropieschätzer, sowie die selben weiteren Verwaltungsvariablen des RNGs genutzt.

### 2.6.1 ssleay_rand_add

Der Zugriff auf den RAND_SSLeay Zustand beim Seeden erfolgt schematisch wie in 2 dargestellt.




md_count[1] zählt, wie oft Datenblöcke in den RNG-Zustand eingepflegt wurden. Ein Datenblock hat eine maximale Länge von MD_DIGEST_LENGTH Bytes.

```
193  static void ssleay_rand_add(const void *buf, int num, double
add)
          {
195       int i,j,k,st_idx;
          long md_c[2];
          unsigned char local_md[MD_DIGEST_LENGTH];
          EVP_MD_CTX m;
          int do_not_lock;
200
          if (!num)
               return;
          ...
```

Im Anschluss wird der exklusive Zugriff auf den Zustand des RNG sichergestellt (z. 220-231). Das Lock CRYPTO_LOCK_RAND wird für die Synchronisierung genutzt. Da diese Variable Prozess-

Global ist, muss überprüft werden, ob der aktuelle Thread das Lock hat. Wenn dies der Fall ist, muss nicht neu gelockt werden (do_not_lock = 1). Dies kann bspw. der Fall sein, wenn ssleay_rand_add() aus ssleay_rand_bytes() heraus aufgerufen wird.

Damit das Locking in md_rand.c funktioniert, muss die registrierte Locking-Funktion einfaches Locking unterstützen, die Unterstützung von rekursiven Locks ist nicht notwendig.

Die Zeilen 232-262 werden durch CRYPTO_LOCK_RAND geschützt ausgeführt. Es werden zunächst Kopien der statischen Variablen in lokalen Variablen angelegt (Z. 232-241, Kopieren von state_index, md_count, und md). Dann wird state_num um die Anzahl Blöcke erhöht, die in den RNG-Zustand eingespeist werden, bis er maximal STATE_SIZE Bytes enthält (Z. 243-255). Die aktuelle Lese/Schreibposition state_index wird ebenfalls erhöht, um den Arbeitsbereich für diesen Seeding-Aufruf zu reservieren. Dabei wird state_buf als Ringpuffer behandelt, und somit state_index auf den Anfang des RNG-Zustands zurückgesetzt, sobald er über das Ende des RNG-Zustands state_buf hinausgeht.

Der Zähler für Seeding-Vorgänge md_count[1] wird um $\lceil num/\text{MD\_DIGEST\_LENGTH} \rceil$ erhöht (Z. 261). Dann wird CRYPTO_LOCK_RAND freigegeben (Z. 263).

In den Zeilen 265-312 erfolgt das Update des RNG-Zustands mit den Seed-Daten. Für jede Runde wird berechnet, welcher Block aus den Eingangsdaten in der aktuellen Runde dem RNG-Zustand hinzugefügt wird (Z. 267-270). Im Normalfall sind dies MD_DIGEST_LENGTH Bytes, in der letzten Runde können es weniger sein, sofern die Anzahl hinzuzufügender Bytes nicht ohne Rest durch MD_DIGEST_LENGTH teilbar ist.

```
266  for (i=0; i<num; i+=MD_DIGEST_LENGTH)
          {
          j=(num-i);
          j=(j > MD_DIGEST_LENGTH)?MD_DIGEST_LENGTH:j;
270           MD_Init(&m);

          MD_Update(&m,local_md,MD_DIGEST_LENGTH);
          k=(st_idx+j)-STATE_SIZE;
          if (k > 0)
275           {
              MD_Update(&m,&(state[st_idx]),j-k);
              MD_Update(&m,&(state[0]),k);
              }
          else
280           MD_Update(&m,&(state[st_idx]),j);

          /* DO NOT REMOVE THE FOLLOWING CALL TO MD_Update()! */
          MD_Update(&m,buf,j);
          ...
291       MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
          MD_Final(&m,local_md);
          md_c[1]++;

295       buf=(const char *)buf + j;
          for (k=0; k<j; k++)
              {
              ...
307           state[st_idx++]^=local_md[k];
              if (st_idx >= STATE_SIZE)
                   st_idx=0;
310           }
          }
```

Abschließend wird der Schreibzähler md_c[1] erhöht (Z. 293) und der Hashwert mittels XOR-Verknüpfung in den RNG-Zustand eingepflegt (Z. 296-310).

Dieser Vorgang wird so lange wiederholt, bis alle übergebenen Entropiedaten in den RNG-Zustand eingepflegt wurden.

Nach dem Einpflegen der Entropie in den Pool wird die globale Entropieschätzung um die ge-

schätzte hinzugefügte Entropie erhöht, sofern die Schätzung nicht bereits größer als die notwendige Entropie (32 Bytes) ist. und der globale Hashwert md mit local_md XOR-verknüpft. Diese beiden Operationen erfolgen wiederum synchronisiert über CRYPTO_LOCK_RAND (Z. 314-325).

Durch die Verkettung der Hashwerte über local_md und md wird erreicht, dass extrahierte Daten von allen vorigen Zuständen des RNG-Zustands abhängig sind.

### 2.6.2 ssleay_rand_seed

```
332  static void ssleay_rand_seed(const void *buf, int num)
          {
          ssleay_rand_add(buf, num, (double)num);
335       }
```

Diese Funktion ist ein Adapter für ssleay_rand_add(), bei der die Entropiemenge gleich der Anzahl der hinzugefügten Bytes gesetzt wird.

### 2.6.3 ssleay_rand_bytes

Die Funktion ssleay_rand_bytes() ist für die Extraktion von Zufallsdaten aus dem RNG-Zustand zuständig.

```
337  static  int  ssleay_rand_bytes(unsigned  char  *buf,  int  num,
int pseudo)
          {
          static volatile int stirred_pool = 0;
340       int i,j,k,st_num,st_idx;
          int num_ceil;
          int ok;
          long md_c[2];
          unsigned char local_md[MD_DIGEST_LENGTH];
345       EVP_MD_CTX m;
          ...
347       pid_t curr_pid = getpid();
          ...
349       int do_stir_pool = 0;
          ...
362       if (num <= 0)
               return 1;
```

**Empfehlung 17:** Wenn eine negative Menge von Zufallsdaten entnommen werden sollen (num<0), wird die Funktion erfolgreich mit Rückgabewert 1 beendet.

**Empfehlung 17:** Auch wenn dieses Verhalten ein Sonderfall des Programmierfehlers ist, dass eine falsche Menge von Zufallsdaten angefordert wird, wäre es hier wünschenswert, durch einen Rückgabewert „-1“ einen Fehler an die aufrufende Stelle zu signalisieren.

Dann wird das CRYPTO_LOCK_RAND genommen (Z. 390-396), um die Zugriffe auf den RNG-Zustand in den Zeilen 390-472 zu synchronisieren.

Sofern der RNG noch nicht initialisiert wurde, erfolgt ein Aufruf von RAND_poll(), um Entropie aus dem Betriebssystem zum RNG-Zustand hinzuzufügen (Z. 398-402). Nach diesem Aufruf ist der RNG initialisiert, unabhängig davon, ob tatsächlich Entropie hinzugefügt werden konnte oder nicht.

Wenn weniger als ENTROPY_NEEDED Bytes Entropie eingespeist wurden, wird die Menge der entnommenen Zufallsdaten vom Entropiezähler abgezogen. Dies wird damit begründet, dass in einem solchen Fall mit Hilfe der entnommenen Entropie der Zustand des RNG rekonstruiert werden kann (Z. 407-425). Wenn der Entropiezähler kleiner „0“ ist, wird er auf „0“ gesetzt.

Um die eingespeiste Entropie auf den kompletten RNG-Zustand zu verteilen wird als nächstes der RNG-Zustand durchmischt. Dies erfolgt dadurch, dass mindestens STATE_SIZE Bytes eines Dummy-Seeds in den RNG-Zustand mittels der Funktion ssleay_rand_add() hinzugefügt wird (Z. 410, 427-450). Wenn einmal ausreichend Entropie im Pool ist, wird das Durchmischen des Pools nicht wiederholt.

```
427  if (do_stir_pool)
          {
          ...
436       int n = STATE_SIZE; /* so that the complete pool gets
accessed */
          while (n > 0)
               {
          ...
442  #define DUMMY_SEED "...................."/* at least MD_DI-
GEST_LENGTH */
               /* Note that the seed does not matter, it's just
that
                 * ssleay_rand_add expects to have something to
hash. */
445            ssleay_rand_add(DUMMY_SEED,       MD_DIGEST_LENGTH,
0.0);
               n -= MD_DIGEST_LENGTH;
               }
          if (ok)
               stirred_pool = 1;
450       }
```

**Empfehlung 18:** Die Variable stirred_pool sollte wie die anderen statischen Variablen global definiert werden und mit der Funktion ssleay_rand_cleanup() zurückgesetzt werden.

Bevor das Lock freigeben wird werden die globalen Variablen kopiert (Z. 452-456), ein Arbeitsbereich des RNG-Zustands reserviert (Z. 458-463) und der Lesezähler md_count[0] erhöht. Dann wird das Lock freigegeben (Z. 468-472).

Das Verfahren für die Extraktion von Entropie ist ähnlich zu dem in Illustration 2 für das Hinzufügen von Entropie dargestellten, mit der grundsätzlichen Ausnahme, dass die maximale Blockgröße nicht der Hashlänge entspricht, sondern nur halb so lang ist.

```
     while (num > 0)
475       {
          /* num_ceil -= MD_DIGEST_LENGTH/2 */
          j=(num >= MD_DIGEST_LENGTH/2)?MD_DIGEST_LENGTH/2:num;
          num-=j;
          MD_Init(&m);
480  #ifndef GETPID_IS_MEANINGLESS
          if (curr_pid) /* just in the first iteration to save
time */
               {
               MD_Update(&m,(unsigned      char*)&curr_pid,sizeof
curr_pid);
               curr_pid = 0;
485            }
     #endif
          MD_Update(&m,local_md,MD_DIGEST_LENGTH);
          MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
          ...
498       MD_Update(&m,buf,j);
          ...
501       k=(st_idx+MD_DIGEST_LENGTH/2)-st_num;
          if (k > 0)
               {
               MD_Update(&m,&(state[st_idx]),MD_DIGEST_LENGTH/2-
k);
505            MD_Update(&m,&(state[0]),k);
               }
          else
               MD_Update(&m,&(state[st_idx]),MD_DIGEST_LENGTH/2);
          MD_Final(&m,local_md);
510
          for (i=0; i<MD_DIGEST_LENGTH/2; i++)
               {
               state[st_idx++]^=local_md[i];
               if (st_idx >= st_num)
515                 st_idx=0;
               if (i < j)
                    *(buf++)=local_md[i+MD_DIGEST_LENGTH/2];
               }
          }
```

In den Zeilen 474-519 wird die Entropie extrahiert. Dazu wird zunächst die Blockgröße für die Extraktion festgelegt. Diese ist normalerweise MD_DIGEST_LENGTH/2, sofern es nicht der letzte Block ist, der kürzer ist, wenn weniger Bytes angefordert wurden (Z. 477). Dann wird die PID (nur beim ersten Block, Z. 480-486), der Hashwert local_md (Z. 487), die Lese- und Schreibzähler md_c (Z. 488), der aktuelle Block des Ausgabepuffer (d.h. eingehende Daten, Z. 498), und der aktuelle

Block des RNG-Zustands gehasht (Z. 501-508). Der Hashwert wird in der Variable local_md gespeichert (Z. 509).

Die erste Hälfte des Hashwerts wird in den RNG-Zustand mittels XOR eingepflegt, während die Zweite Hälfte im Ausgabepuffer gespeichert wird (Z. 511-519). Dieser Vorgang wird so lange wiederholt, bis die angeforderte Menge von Zufallsdaten extrahiert wurde. Die Verwendung der Hashfunktion bei der Extraktion verhindert, dass ein Beobachter anhand der aus dem RNG-Zustand entnommenen Daten auf dessen Zustand schließen kann.

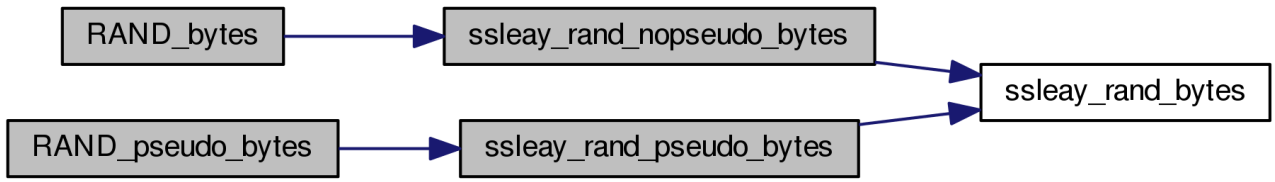
Das Hinzufügen des alten Inhalts des zu füllenden Puffers stellt kein Sicherheitsproblem dar, da die Daten mit der Hashfunktion gemischt werden. Aufgrund möglicher Kollisionen in Hashfunktionen besitzt der interne Zustand im Worst Case eine Min-Entropie von $\sqrt{\text{ENTROPY\_NEEDED}} \geq 16\,\text{Bytes}$ , so dass es dem Angreifer auch nicht möglich ist, mittels geeignetem Raten auf den internen Zustand zu schließen.

Nach der Entnahme der Entropie wird nochmal der letzt Hashwert local_md und der globale Hashwert md gehasht und als neuer globaler Hashwert in md gespeichert. Dieses Update erfolgt über CRYPTO_LOCK_RAND synchronisiert (Z. 521-535).

Abschließend wird der Rückgabewert erzeugt, wobei dieser davon abhängig ist, ob kryptografisch starke oder schwache Zufallsdaten angefordert und geliefert wurden (Z. 536-546).

### 2.6.4 ssleay_rand_pseudo_bytes und ssleay_rand_nopseudo_bytes()

Diese Funktionen sind triviale Adapter-Funktionen, um das Interface von ssleay_rand_bytes() auf das generische OpenSSL-Interface abzubilden.




### 2.6.5 ssleay_rand_cleanup

Diese Funktion setzt den Zustand des RNG zurück. Es werden die in Abs. 2.6.1 genannten Zustandsvariablen auf 0 gesetzt.

### 2.6.6 ssleay_rand_status

Diese Funktion stellt fest, ob der RNG genutzt werden kann. Sofern der RNG nicht initialisiert ist, nimmt sie auch die Initialisierung vor, sofern ausreichend Entropiequellen vorhanden sind. Hierdurch wird der Zustand des RNG verändert. Wenn der RNG entweder vor dem Aufruf ausreichend geseedet wurde oder während des Aufrufs erfolgreich geseedet werden konnte, gibt die Funktion den Wert „1“ zurück, ansonsten den Wert „0“.

### 2.6.7 Entropiequellen

In rand_unix.c ist die Funktion RAND_poll() implementiert, die den RNG mit betriebssystemspezifischer Entropie beim Initialisieren des RNG seedet. Es existieren keine Funktionen um die genutzen Entropiequellen festzustellen.

Als Entropiequellen werden hier die folgenden Daten genutzt:

| Quelle | Entropieschätzung | Bemerkung |
|---|---|---|
| /dev/urandom | 100 Prozent | Entropiequelle des OS |
| /dev/random | 100 Prozent | Entropiequelle des OS |
| getuid() | 0 | UID des Prozess |
| time() | 0 | Zeit |
| ssleay_rand_add() | vom Anwender | Pflegt Entropie vom Benutzer ein, mit dessen Entropieschätzung |

*Tabelle 2.1: Entropiequellen und Entropieschätzung*

Aus den Entropiequellen des Betriebssystems werden bis zu ENTROPY_NEEDED Bytes gelesen. Sobald die benötigte Entropie gesammelt wurde, werden die darauf folgenden Quellen übersprungen. Der Aufruf der Quellen erfolgt in der Reihenfolge wie in Tabelle Tabelle 2.1 dargestellt. Die so erlangte Entropie wird mit RAND_add() dem RNG-Zustand hinzugefügt. Der Programmcode zum Hinzufügen der Entropie aus den Geräte-Dateien verfährt wie folgt:

1. Lesen von maximal 32 Bytes oder maximal so lange, bis das Timeout von 10ms erreicht wurde
2. Hinzufügen der gelesenen Bytes (32 oder weniger, falls Timeout erreicht wurde) zum RNG-Zustand mittels RAND_add(), wobei die Anzahl gelesener Bytes der Schätzwert ist.

Der Entropiezähler ist nicht direkt über ein von OpenSSL angebotenes Interface ansprechbar, sondern es kann lediglich mit der Funktion RAND_status() festgestellt werden, ob mehr oder weniger als 32 Bytes Entropie eingeflossen sind.

**Bemerkung 19:** Als erste Entropiequelle wird /dev/urandom gelesen, die unter Linux aber nicht unbedingt kryptografisch sichere Zufallsdaten liefert. Da /dev/urandom nicht blockiert und soviel Da-

ten liefert wie angefordert, wird /dev/random in typischen Setups überhaupt nicht verwendet.

**Empfehlung 19:** Es sollte als erste Entropiequelle /dev/random gelesen und/oder /dev/urandom im Quelltext komplett deaktiviert werden. Auf /dev/random sollte außerdem nicht verzichtet werden. Alternativ kann der RNG manuell mit der Funktion RAND_add() mit Entropie aus einer guten Zufallsquelle geseedet werden.

**Bemerkung 20:** getpid(), getuid() und time(), deren Typen plattformabhängig sind und damit unterschiedliche Größe haben können, werden erst in einen unsigned long konvertiert, bevor sie dem Pool hinzugefügt werden. Dies kann dazu führen, dass nur insignifikante Bits in den RNG-Zustand eingemischt werden, wenn die betroffene Struktur größer ist als 8 Bytes. Wenn die Struktur kleiner ist als 8 Bytes, kann es Zugriffe auf uninitialisierten Speicher nach sich ziehen.

**Empfehlung 20:** Die genannten Werte sollten mit ihrer tatsächlichen Größe dem RNG-Zustand hinzugefügt werden.

**Bemerkung 21:** Es ist wenig über die Qualität des Entropy-Gathering-Daemon bekannt.

**Empfehlung 21:** Da die Linux-Kernel-Entropiequellen weitläufig genutzt und dokumentiert sind, sollten ausschließlich Kernel-Entropiquelle genutzt werden und die Unterstützung für den Entropy Gathering Daemon für die Linux-Plattform deaktiviert werden.

**Bemerkung 22:** Der Entropy-Gathering-Daemon ist standardmäßig aktiviert, wenn er verfügbar ist.

**Empfehlung 22:** Unterstützung für den EGD sollte in der Datei e_os.h deaktiviert werden, indem die Definition DEVRAND_EGD zu einer leeren liste gesetzt wird (e_os.h, Z. 88 oder per Aufruf des Konfigurations-Scripts mit „./config -DDEVRANDOM_EGD=NULL“, s. [AP3], Abs. 3.2.9).

**Bemerkung 23:** Es ist für einen Programmierer nicht möglich festzustellen, aus welchen Entropiequellen die Entropie beim Seeding stammte.

**Empfehlung 23:** Nutzer von OpenSSL sollten den RNG explizit mit Daten aus einer zuverlässigen Entropiequellen seeden.

## 2.7 Veränderungen im aktuellen OpenSSL und LibreSSL

Veränderungen OpenSSL, Stand: 17.09.2014

- 2014-01-11: Daten von Intel rdrand werden bei der Entnahme von Entropie in ssleay_rand_bytes() hinzugefügt. Der Schätzwert für die Entropie beträgt 0.
- 2014-04-07: ssleay_rand_add gibt einen Fehler zurück, wenn die Funktion mit 0 Bytes Daten als Argument aufgerufen wird.
- 2014-06-07: Korrektur eines Bugs, der die Entropiedatei ~/.rnd mit falschen Rechten erstellt.

Veränderungen LibreSSL, Stand: 17.09.2014

- In LibreSSL wurde der RAND_SSLeay komplett entfernt und durch Betriebssystem-Entropiequellen ersetzt.

## 2.8 Übersicht: Lebenszyklus des RNGs

**Initialisierung**

Die Initialisierung des RNGs erfolgt implizit in ssleay_rand_bytes().

**Extraktion**

Extraktion von Entropie erfolgt mittels ssleay_rand_bytes().

**Seeding**

Seeding erfolgt mittels ssleay_rand_add().

**Entropiequellen**

Entropie wird implizit in ssleay_rand_bytes() über RAND_poll() hinzugefügt, sowie manuell durch den Anwender.

**Beenden**

Die Funktion ssleay_rand_cleanup()/RAND_cleanup() löscht den Zustand des RNGs.

**Sicherheitskritische Daten**

- Die statischen Variablen state und md mit dem Zustand des RNGs werden sicher überschrieben, sofern ssleay_rand_cleanup() aufgerufen wird.
- Der temporäre Speicher für die Hashwertberechnung beim Seeding und der Entropieextraktion wird nach Gebrauch durch EVP_MD_CTX_cleanup() gelöscht. In EVP_MD_CTX_cleanup() wird der Speicher für den Hashwert mit OPENSSL_cleanse() überschrieben.

**Parameter des RNGs**

| Parameter | Quelldatei/Variablenname | Wert |
|---|---|---|
| Hashfunktion | rand_lcl.h | SHA1 |
| Länge des Hashes | rand_lcl.h | 20 Bytes |
| Entropie für Initialisierung | rand_lcl.h:ENROPY_NEEDED | 32 Bytes |
| Größe des RNG-Zustands | md_rand.c:STATE_SIZE | 1023 Bytes |

### 2.8.1 Sichere Verwendung des RNGs

Aufgrund der dargestellten Probleme, insbesondere der Reihenfolge beim impliziten Seeding mit /dev/urandom, sollte ein Programmierer sich nicht auf das implizite Seeding verlassen, sondern explizit mittels RAND_add() Entropie in den RNG einbringen.

1. Daten aus guter Zufallsquelle (z.B. NTG.1) lesen. Es muss sichergestellt werden, dass Daten mit mindestens 32 Bytes Entropie gelesen werden
2. Seeden des RNGs mit den so entnommenen Daten.
3. Erst nun dürfen kryptografische Operationen durchgeführt werden.

Dieser Vorgang muss nach jedem Erzeugen von neuen Prozessen erfolgen. Um darüber hinaus sicher zu stellen, dass Daten aus dem Zustand des RNGs nicht doppelt verwendet werden, dürfen Zufallsdaten nur von einem Thread entnommen werden.

# 3 Verwendung von RNGs in OpenSSL

## 3.1 Nutzung des Standard-RNGs in OpenSSL

Auf den SSLeay-RNG wird von OpenSSL nicht direkt zugegriffen. Zugriff auf die RNG-Funktionen erfolgen ausschließlich über das API-Interface. Die Nutzung der Funktionen ist in Tabelle Tabelle 3.1 dargestellt.

Untersucht werden:

- Behandlung des Rückgabewerts (für RAND_status(), RAND_bytes(), RAND_pseudo_bytes())
- Bewertung der Entropieschätzung, sofern sie nicht 0 ist (für RAND_add(), RAND_seed())
- Tests, TLS-SRP, SSLv2 und DTLS werden in dieser Analyse nicht betrachtet, werden als Referenz aber dennoch aufgeführt. Eine detaillierte Untersuchung der Programmlogik konnte im Rahmen der Arbeiten nicht durchgeführt werden.

Auf eine Auflistung der einzelnen Funktion wurde zugunsten der Dateinamen verzichtet, da sowohl das für die Bewertung notwendige Verständnis nur im Kontext der gesamten Datei erfolgen kann, als auch die Entropiefunktionen in den jeweiligen Dateien nur selten aufgerufen werden und daher auch ohne Funktionsnamen schnell auffindbar sind.

Die Analyse betrachtet sowohl die generelle Verwendung der RAND_bytes() und RAND_pseudo_bytes()-Funktionen gemäß ihrer Spezifikation (kryptografisch starke bzw. pseudo-Zufallsdaten), als auch die konkrete Implementierung durch den Standard-RNG.

| Datei | add[1] | seed[2] | status[3] | bytes[4] | pseudo[5] | cleanup[6] | Bemerkung |
|---|---|---|---|---|---|---|---|
| apps/app_rand.c | | | X | | | | ok |
| apps/apps.h | | | | | | X | ok |
| apps/enc.c | | | | | X | | IV |
| apps/genrsa.c | | | X | | | | ok |
| apps/passwd.c | | | | | X | | Salt |
| apps/rand.c | | | | X | | | ok |
| apps/s_cb.c | | | | X | | | Fehler: Test auf „! RAND_bytes()“[7] |
| apps/s_client.c | | | X | | | | ok |
| apps/speed.c | | X | X | | X | X | Fehler: Fehlercode von RAND_pseudo_bytes() ignoriert |
| apps/s_server.c | | | X | | X | | Fehler: Fehlercode von RAND_pseudo_bytes() ignoriert |
| apps/ts.c | | | | X | | | ok, nonce |
| crypto/asn1/asn_mime.c | | | | | X | | Fehler: Fehlercode ignoriert |
| crypto/asn1/p5_pbe.c | | | | | X | | ok |
| crypto/asn1/p5_pbev2.c | | | | | X | | ok |
| crypto/bio/bf_nbio.c | | | | | X | | Fehler: Fehlercode ignoriert |
| crypto/bn/bn.h | | | | X | | | Fehler: Fehlercode ignoriert |
| crypto/bn/bn_rand.c | X | | | X | X | | Fehler: Fehlercode von RAND_pseudo_bytes() ignoriert |
| crypto/bn/bntest.c | | X | | | | | Test |

1 Funktion RAND_add()
2 Funktion RAND_seed()
3 Funktion RAND_status()
4 Funktion RAND_bytes()
5 Funktion RAND_pseudo_bytes()
6 Funktion RAND_cleanup()
7 RAND_bytes kann im Fehlerfall -1 zurückgeben, was vom Code nicht korrekt behandelt wird

| Datei | add | seed | status | bytes | pseudo | cleanup | Bemerkung |
|---|---|---|---|---|---|---|---|
| crypto/bn/divtest.c | | | | | X | | Fehler: Fehlercode ignoriert |
| crypto/bn/expspeed.c | | X | X | | | | Test |
| crypto/bn/exptest.c | | X | | X | | | Test |
| crypto/dh/dhtest.c | | X | | | | | Test |
| crypto/dsa/dsa_asn1.c | | X | | | | | Fehler: Seeding mit Hash einer zu signierenden Nachricht |
| crypto/dsa/dsa_gen.c | | | | | X | | Fehler: Schlüsselerzeugung mit schwachen Zufallsdaten; Fehlercode ignoriert |
| crypto/ecdh/ecdhtest.c | | X | | | | | Test |
| crypto/ecdsa/ecdsatest.c | | X | | | X | | Test |
| crypto/ecdsa/ecs_sign.c | | X | | | | | Fehler: Seeding mit Hash einer zu signierenden Nachricht |
| crypto/ec/ectest.c | | X | | | | | Test |
| crypto/evp/bio_ok.c | | | | | X | | Fehler: Fehlercode ignoriert |
| crypto/evp/e_aes.c | | | | X | | | IV |
| crypto/evp/evp_pkey.c | X | | | | | | ok[8] |
| crypto/evp/p_seal.c | | | | X | | | Fehler: Fehlercode ignoriert |
| crypto/ocsp/ocsp_ext.c | | | | | X | | Fehler: Fehlercode ignoriert |
| crypto/pem/pem_lib.c | X | | | | X | | nonce |
| crypto/pem/pvkfmt.c | | | | X | | | ok, salt |

8 Es wird der RSA private Key zum RNG-Zustand mit der Entropieschätzung 0 hinzugefügt. Im Gegensatz zu http://opensslrampage.org/post/83007010531/well-even-if-time-isnt-random-your-rsa-private-key sehen wir darin kein Problem, da der Zustand des RNG bei korrekter Nutzung nicht von einem Angreifer rekonstruiert werden kann, sofern dieser nicht privilegierte Rechte auf den OpenSSL besitzt. Das gleiche gilt, falls der RNG von einer engine zur Verfügung gestellt wird: Dieser sollte in jedem Fall vertrauenswürdig sein, da ansonsten die Sicherheit der gesamten Sicherheitsarchitektur nicht gewährleistet werden kann.

| Datei | add | seed | status | bytes | pseudo | cleanup | Bemerkung |
|---|---|---|---|---|---|---|---|
| crypto/pkcs12/p12_mutl.c | | | | | X | | salt |
| crypto/pkcs1/pk7_doit.c | | | | | X | | IV |
| crypto/rsa/rsa_crpt.c | X | | X | | | | ok[9] |
| crypto/rsa/rsa_oaep.c | | | | X | | | ok |
| crypto/rsa/rsa_pk1.c | | | | X | | | ok, padding |
| crypto/rsa/pss.c | | | | X | | | ok, salt |
| crypto/rsa/rsa_ssl.c | | | | X | | | ok, padding |
| crypto/rsa/rsa_test.c | | X | | | | | Test |
| crypto/threads/mttest.c | | X | | | | | Test |
| demos/easy_tls/easy-tls.c | | X | | X | | | Fehler: RAND_bytes() Fehlercode ignoriert |
| ssl/d1_both.c | | | | | X | | DTLS |
| ssl/d1_clnt.c | X | | | X | | | DTLS |
| ssl/d1_enc.c | | | | X | | | DTLS |
| ssl/d1_pkt.c | | | | | X | | DTLS |
| ssl/d1_srvr.c | X | | | | X | | DTLS |
| ssl/s23_clnt.c | X | | | | X | | ok, nutzt RAND_pseudo_bytes() äquivalent zu RAND_bytes() in SSL-HELLO |
| ssl/s23_srvr.c | X | | | | | | ok |
| ssl/s2_clnt.c | X | | | X | X | | SSLv2 |
| ssl/s2_srvr.c | X | | | | X | | SSLv2 |
| ssl/s3_clnt.c | X | | | X | | | Fehler: RAND_bytes() Fehlercode wird beim Erzeugen des Premaster Secrets ignoriert[10] |

9 Es wird der RSA private Key zum RNG-Zustand mit der Entropieschätzung 0 hinzugefügt. Siehe Fußnote 8.

10 Dieser Fehler ist im Kontext der Verwendung von OpenSSL als Bibliothek durch Dritte besonders kritisch, da dieser Code sicher häufig als Basis oder Beispielcode für neue Programmentwicklungen genutzt werden und der Fehler sich dadurch weiter verbreitet.

| Datei | add | seed | status | bytes | pseudo | cleanup | Bemerkung |
|---|---|---|---|---|---|---|---|
| ssl/s3_srvr.c | X | | | | X | | nutzt RAND_pseudo_bytes() äquivalent zu RAND_bytes(), ignoriert Rückgabewert von RAND_pseudo_bytes() |
| ssl/ssl_lib.c | | | | X | X | | ok, nutzt RAND_pseudo_bytes() äquivalent zu RAND_bytes() |
| ssl/ssl_sess.c | | | | | X | | session ID |
| ssl/ssltest.c | | X | | | | | Test |
| ssl/t1_enc.c | | | | X | | | ok, IV |
| ssl/t1_lib.c | | | | | X | | Padding, Fehlercode ignoriert |
| ssl/tls_srp.c | | | | X | | | TLS-SRP |
| test/igetest.c | | | | | X | | Test |

*Tabelle 3.1: Verwendung des RNG innerhalb von OpenSSL*

### 3.1.1 Analyse

Die Nutzung von RAND_pseudo_bytes() kann im Worst-Case dazu führen, dass der Zufallszahlengenerator ohne Seeding mit Entropie benutzt wird.

**Bemerkung 24:** Mittels RAND_pseudo_bytes() entnommene Zufallsdaten können vorhersagbar sein.

**Empfehlung 24:** Es muss sichergestellt werden, dass vor dem Aufruf der Funktion RAND_pseudo_bytes() der RNG-Zustand mit ausreichend Entropie geseedet wurde. Hierfür empfiehlt sich, beim Programmstart den RNG-Zustand zu seeden und dann mittels RAND_status() zu überprüfen, ob der RNG für kryptografische Operationen verwendet werden kann (Rückgabewert = 1). Ansonsten sollte sich das Programm mit einer Fehlermeldung beenden.

**Bemerkung 25:** In crypto/dsa/dsa_gen.c wird RAND_pseudo_bytes() für die Erzeugung eines Schlüssels verwendet.

**Empfehlung 25:** Für Schlüsselerzeugung sollte immer RAND_bytes() genutzt werden. Die Verwendung zur Erzeugung von Schlüsseln in crypto/dsa/dsa_gen.c muss zur Verwendung von

RAND_bytes() geändert werden.

### 3.1.2 Fehlerklasse: Ignorieren von Fehlercodes

**Bemerkung 26:** Wenn ein Fehlercode bei der Entnahme von kryptografisch starker Entropie ignoriert wird, können Sicherheitsanforderungen nicht mehr erfüllt werden. Am offensichtlichsten ist dies bei der Schlüsselerzeugung, aber auch bei der Erzeugung von Initialisierungsvektoren wie in Zeile 3437 von ssl/s3_srvr.c darf der Rückgabewert nicht ignoriert werden.

**Empfehlung 26:** Der Fehlercode der Funktion RAND_bytes() darf nicht ignoriert werden

**Bemerkung 27:** Wenn ein Fehlercode bei der Entnahme von Zufallsdaten ignoriert wird, kann es sein, dass der Buffer für die Aufnahme der Zufallsdaten noch den gleichen Inhalt wie vor dem Aufruf besitzt. Wenn diese Daten anschließend an den Angreifer geschickt werden, kann dieser unbefugten Zugriff auf Daten bekommen, die von vorherigen Operationen an der Stelle steht, wo die neuen Zufallsdaten geschrieben werden sollten.

**Empfehlung 27:** Der Fehlercode der Funktion RAND_pseudo_bytes() darf nicht ignoriert werden.

### 3.1.3 Fehlerklasse: Seeding mit Hash einer zu signierenden Nachricht

**Bemerkung 28:** Beim Seeding des RNG mit dem Hashwert der zu signierenden Nachricht kann die Entropieschätzung zu optimistisch sein. Dies liegt darin begründet, dass die Entropieschätzung beim Seeding mit der Funktion RAND_seed() der Anzahl von Bytes entspricht, die dem RNG-Zustand hinzugefügt werden. Bei der Hash-Funktion SHA-256 sind dies somit 32 Bytes. Auch wenn keine weiteren Entropiequellen genutzt werden können reicht diese Menge von Entropie aus, damit der RNG kryptografisch sichere Zufallsdaten liefert – was aber nicht der tatsächlich verfügbaren Entropie entspricht. Wenn ein Angreifer nun die signierte Nachricht erraten kann, kann er daraus den Zustand des RNG-Zustands rekonstruieren. Dies kann insbesondere zu Problemen führen, wenn ein Selbsttest der Signatur durchgeführt wird bevor der RNG korrekt geseedet wurde.

**Empfehlung 28:** Beim Hinzufügen von Daten zum RNG-Zustand aus Quellen, bei denen nicht sichergestellt werden kann, wie groß deren Entropie ist, sollte die Entropie mit 0 geschätzt werden.

### 3.1.4 Muster: Verwenden von RAND_pseudo_bytes() äquivalent zu RAND_bytes()

An einigen Stellen wird die Fehlerbehandlung von RAND_pseudo_bytes() so durchgeführt wie bei RAND_bytes(). Dies hat zur Folge, dass RAND_pseudo_bytes() sich genau so verhält wie RAND_bytes(), und einen Fehler zurückgibt, selbst wenn pseudozufällige Daten zurückgegeben werden. Beachtlich ist hier ein Kommentar in ssl/s3_srvr.c, Z. 2250, wo RAND_pseudo_bytes() wie RAND_bytes() genutzt wird, aber offensichtlich nicht so genutzt werden darf („should be RAND_bytes, but we cannot work around a failure“).

Weitere Informationen zur Problematik der Rückgabewerte wurden bereits in Abschnitt 1.1.1 dargestellt.

## 3.2 Erzeugung von zufälligen BigNum-Zahlen

Für die Erzeugung von zufälligen BigNum-Zahlen existieren zwei Klassen von Funktionen. Für nicht kryptografisch starke Zufallsdaten gibt es die Funktionen BN_pseudo_rand sowie BN_pseudo_rand_range, und für kryptografisch starke Zufallsdaten werden BN_rand() und BN_rand_range() bereitgestellt.

BN_pseudo_rand_range() und BN_pseudo_rand() werden (abgesehen von Tests) in crypto/bn/bn_prime.c in einem Primzahltest, in apps/apps.c in bestimmten Fällen für die Erzeugung einer Seriennummer für ein Zertifikat sowie in bn_sqrt.c für die Berechnung einer Wurzel in einem Restklassenring genutzt. Die Verwendung dieser Zufallsdaten ist in allen Fällen angemessen.

In allen anderen Fällen (Schlüsselerzeugung DSA, Erzeugen von Blinding-Parametern, Erzeugung von k für DSA- und ECDSA-Signaturerstellung, ECC Schlüsselerzeugung, DH-Schlüsselerzeugung, Erzeugung von Primzahlen) wird der starke RNG BN_rand_range() bzw. BN_rand() genutzt.

BN_rand_range() und BN_pseudo_rand_range() nutzen die Funktion bn_rand_range(), die anhand eines Parameters intern die Funktion BN_rand() oder BN_pseudo_rand nutzt. BN_rand() und BN_pseudo_rand() nutzen intern die Funktion bnrand(), die wiederum RAND_bytes() oder RAND_pseudo_bytes() aufruft.

Die Funktionen sind in crypto/bn/bn_rand.c implementiert.

```
static int bnrand(int pseudorand, BIGNUM *rnd, int bits, int top,
int bottom)
static int bn_rand_range(int pseudo, BIGNUM *r, const BIGNUM *ran-
ge)
```

### *3.2.1.1 bnrand()*

Der Parameter „pseudorand“ gibt an, ob die Zufallsdaten mit der Funktion RAND_pseudo_bytes() oder RAND_bytes() erzeugt werden sollen. Es wird ein Puffer mit „bits“ Zufallsbits gefüllt, und diese Bitfolge werden dann mit der Funktion „BN_bin2bn“ zu einer entsprechenden Ganzzahl umgewandelt.

Anhand des Parameters „top“ wird entschieden, ob die führenden Bits gesetzt sein müssen oder auch ungesetzt bleiben dürfen. Wenn der Parameter „bottom“ gesetzt ist, wird das geringwertigste Bit gesetzt.

| top | Wert der beiden MSBs |
|---|---|
| -1 | ??[11] |
| 0 | 1? |
| 1 | 11 |

#### 3.2.1.2 bn_rand_range()

bn_rand_range() erzeugt Zufallszahlen in einem Intervall von [0, …, r]. Es gibt zwei Algorithmen für die Zufallszahlenerzeugung:

Fall A: range = 100???...???

1. Lese Zufallszahl r der Länge $\lceil \log_2(range)+1 \rceil$
2. Wenn $0 \le r < 3 \cdot range$ : return $r \bmod range$ , sonst wiederhole Schritt 1

Fall B: Sonst:

1. Lese Zufallszahl r der Länge $\lceil \log_2(range) \rceil$ mittels bnrand()
2. Wenn $0 \le r < range$ : return r, sonst wiederhole Schritt 1

Beide Verfahren erzeugen gleichverteilte Zufallszahlen r, $0 \le r < range$ . Das Vorgehen im Fall B entspricht [TR021021], B.4. Verfahren 1. Das Verfahren in Fall B ist nicht von der [TR021021] abgedeckt.

## 3.3 Nicht-Verwendung des Standard-RNG durch Kryptoroutinen

Der Standard-RNG wird lediglich in den Tests für die ECDSA-Funktionen ersetzt. Dieses Vorgehen stellt kein Problem dar.

An allen anderen Stellen wird der Standard-RNG verwendet.

## 3.4 Auflistung und Kurzbeschreibung weiterer RNGs

Zusätzliche RNGs werden von optionalen Engines bereitgestellt. Die Registrierung, Verwaltung und der Aufruf der RNG-Engines erfolgt über die in crypto/engine/tb_rand.c bereitgestellten Funktionen. Es können gleichzeitig mehrere Engines verfügbar sein. Wenn eine Engine für den RNG konfiguriert wurde, werden alle Aufrufe an den OpenSSL-RNG von dieser Engine erfüllt.

11 „0“ und „1“ stehen für erzwungene Werte, ein „?“ steht für einen zufälligen Wert, der mit dem RNG erzeugt wird.

| Funktion | Kurzbeschreibung |
|---|---|
| ENGINE_register_RAND() | Registrierung eines RNGs |
| ENGINE_unregister_RAND() | Deregistrierung eines RNGs |
| ENGINE_register_all_RAND() | Registriert die RNG-Methoden aller geladenen Engines |
| ENGINE_unregister_all_RAND() | Deregistriert die RNG-Methoden aller geladenen Engines |
| ENGINE_set_default_RAND() | Setzen des Standard-RNGs |
| ENGINE_get_default_RAND() | Abfrage des Standard-RNGs |
| ENGINE_set_RAND() | Setzen des RNGs einer bestimmten Engine |
| ENGINE_get_RAND() | Abfrage des RNGs einer bestimmten Engine |

### 3.4.1 Intel RDRAND

**Definitionsdatei**: crypto/engine/eng_rdrand.c

Der Intel RDRAND RNG ist ein dünner Abstraktionslayer für den in aktuellen Intel-CPUs über die RDRAND-Instruktion verfügbaren Hardware-RNG. Bei der Nutzung dieser Engine wird der RNG ohne weitere Filterung genutzt. Die in neuen CPUs verfügbare Seeding-Funktion RDSEED des Intel RNG wird nicht unterstützt.

**Bemerkung 29:** Die Integration eines physikalischen RNGs in die CPU ist grundsätzlich eine sinnvolle Maßnahme. Allerdings ist die von Intel bereitgestellte Implementierung nicht auditierbar. Eine Provable-Security-Analyse findet Schwächen bezüglich der Forward- und Backward-Security (http://eprint.iacr.org/2014/504).

**Empfehlung 29:** Die RDRAND-Engine sollte nicht für kryptografisch starke Zufallsdaten genutzt werden.

### 3.4.2 VIA Padlock Engine

**Definitionsdatei**: engines/e_padlock.c

Implementierung für die in VIA-Prozessoren enthaltene PadLock-Engine. Die Seeding-Funktion ist nicht implementiert.

**Bemerkung 30:** Das Interface für rand_pseudo_bytes() ist nicht konform zur OpenSSL-Spezifikation implementiert: So wird der Wert „0“ zurück gegeben, wenn die Entropieerzeugung in der Hardware-Engine fehlschlägt.

**Empfehlung 30:** Die rand_pseudo_bytes()-Funktion der VIA Padlock Engine sollte nicht genutzt werden.

### 3.4.3 Cluster Labs

**Definitionsdatei**: demos/engines/cluster_labs/hw_cluster_labs.c

Implementierung für die Cluster-Labs CRP310 und CRP410 SSL-Beschleuniger[12]. Die Seeding-Funktion ist nicht implementiert.

**Bemerkung 31:** Das Interface für rand_pseudo_bytes() ist nicht konform zur OpenSSL-Spezifikation implementiert: So wird der Wert „0“ zurück gegeben, wenn die Entropieerzeugung in der Hardware-Engine fehlschlägt.

**Empfehlung 31:** Die rand_pseudo_bytes()-Funktion der Cluster Labs Engine sollte nicht genutzt werden.

**Bemerkung 32:** Die Firma Cluster-Labs ist seit 2008 insolvent.

**Empfehlung 32:** Die Cluster-Labs Engine sollte nicht genutzt werden, da der Support der notwendingen Hardware nicht mehr sichergestellt werden kann.

### 3.4.4 IBM CA

**Definitionsdatei**: demos/engines/ibmca/hw_ibmca.c

Implementierung für den „IBM Crypto Adapter“. Die Seeding-Funktion ist nicht implementiert.

**Bemerkung 33:** Das Interface für rand_pseudo_bytes() ist nicht konform zur OpenSSL-Spezifikation implementiert: So wird der Wert „0“ zurück gegeben, wenn die Entropieerzeugung in der Hardware-Engine fehlschlägt.

**Empfehlung 33:** Die rand_pseudo_bytes()-Funktion der IBM CA Engine sollte nicht genutzt werden.

### 3.4.5 Zencod Engine

**Definitionsdatei**: demos/engines/zencod/hw_zencod.c

Implementierung für das „Zencod Zenbridge“ HSM. Die Seeding-Funktion ist nicht implementiert.

**Bemerkung 34:** Das Interface für rand_pseudo_bytes() ist nicht konform zur OpenSSL-Spezifikation implementiert: So wird der Wert „0“ zurück gegeben, wenn die Entropieerzeugung in der Hardware-Engine fehlschlägt.

**Empfehlung 34:** Die rand_pseudo_bytes()-Funktion der Zencod Engine sollte nicht genutzt werden.

12 http://rt.openssl.org/Ticket/Display.html?id=218

### 3.4.6 CryptoSwift Engine

**Definitionsdatei**: engines/e_cswift.c

Implementierung für das „CryptoSwift“ Crypto-Modul. Die Seeding-Funktion ist nicht implementiert.

**Bemerkung 35:** Das Interface für rand_pseudo_bytes() ist nicht konform zur OpenSSL-Spezifikation implementiert: So wird der Wert „0“ zurück gegeben, wenn die Entropieerzeugung in der Hardware-Engine fehlschlägt.

**Empfehlung 35:** Die rand_pseudo_bytes()-Funktion der CryptoSwift Engine sollte nicht genutzt werden.

**Bemerkung 36:** Die Firma Rainbow Technologies, die das CryptoSwift-Modul vertrieben hat, scheint nicht mehr zu existieren.

**Empfehlung 36:** Die CryptoSwift Engine sollte nicht genutzt werden, da der Support der notwendingen Hardware nicht mehr sichergestellt werden kann.

### 3.4.7 CHIL Engine

**Definitionsdatei**: engines/e_chil.c

Implementierung für nCipher CHIL HSM. Die Seeding-Funktion ist nicht implementiert.

**Bemerkung 37:** Das Interface für rand_pseudo_bytes() ist nicht konform zur OpenSSL-Spezifikation implementiert: So wird der Wert „0“ zurück gegeben, wenn die Entropieerzeugung in der Hardware-Engine fehlschlägt.

**Empfehlung 37:** Die rand_pseudo_bytes()-Funktion der CHIL Engine sollte nicht genutzt werden.

### 3.4.8 AEP SureWare Engine

**Definitionsdatei**: engines/e_sureware.c

Implementierung für AEP SureWare HSMs. Die Implementierung ist lediglich ein Wrapper um die SureWare-Bibliothek. Rückgabewerte werden unverändert weitergegeben.

**Bemerkung 38:** Die Spezifikation der Bibliothek ist nicht verfügbar. Daher ist eine Bewertung der SureWare-Engine nicht möglich.

**Empfehlung 38:** Vor der Verwendung der SureWare-Engine sollte ein Review der Engine und der Sureware-Bibliothek erfolgen.

### 3.4.9 AEP Engine

**Definitionsdatei**: engines/e_aep.c

Implementierung für AEP HSMs. Die Seeding-Funktion ist nicht implementiert.

**Bemerkung 39:** Das Interface für rand_pseudo_bytes() ist nicht konform zur OpenSSL-Spezifikation implementiert: So wird der Wert „0“ zurück gegeben, wenn die Entropieerzeugung in der Hardware-Engine fehlschlägt.

**Empfehlung 39:** Die rand_pseudo_bytes()-Funktion der AEP Engine sollte nicht genutzt werden.

### 3.4.10 IBM 4758 Engine

**Definitionsdatei**: engines/e_4758cca.c

Implementierung für IBM 4758 HSMs. Die Seeding-Funktion ist nicht implementiert.

**Bemerkung 40:** Das Interface für rand_pseudo_bytes() ist nicht konform zur OpenSSL-Spezifikation implementiert: So wird der Wert „0“ zurück gegeben, wenn die Entropieerzeugung in der Hardware-Engine fehlschlägt.

**Empfehlung 40:** Die rand_pseudo_bytes()-Funktion der IBM 4758 Engine sollte nicht genutzt werden.

### 3.4.11 ECDSA Dummy RAND

**Definitionsdatei**: crypto/ecdsa/ecdsatest.c

Für den ECDSA-Test werden deterministische Zahlen benötigt. Dafür wird in diesem Test ein deterministischer Zufallszahlengenerator implementiert, der acht verschiedene Werte liefert. Für den Test ist diese Vorgehensweise vertretbar.

### 3.4.12 FIPS RNG

OpenSSL hat einen Zufallszahlengenerator für die FIPS-Variante. Dieser wird in einer speziellen Quelltext-Version verteilt und ist nicht Teil des von OpenSSL 1.0.1g.

# 4 Callgraph RNG-Funktionen