باسمه تعالی

نحوه ذخیره امن رمزهای عبور

فهرست مطالب

۱ مقدمه

یکی از علل به‌وجودآمدن آسیب‌پذیری در برنامه‌ها، خطاهای رایج برنامه‌نویسی می‌باشد. از مهمترین این خطاها می‌توان به نحوه ذخیره‌سازی رمزهای عبور مربوط به برنامه‌های مختلف اشاره نمود. متخصصان امنیت توصیه می‌کنند که به‌جای ذخیره رمز عبور، معادل درهم (hash) آن ذخیره گردد. روش‌های درهم‌سازی موجود بسیار متنوع بوده و به‌طور پیوسته روش‌های جدید معرفی می‌گردند. در این مستند جدیدترین روش‌های مختلف درهم‌سازی بررسی شده و روش‌های برتر معرفی شده‌اند. بدین منظور ابتدا کتابخانه sodium معرفی شده که حاوی تمامی توابع جدید مورد نیاز بوده و استفاده از آن توسط متخصصان توصیه شده است. در بخش دوم گزارش، نحوه ذخیره امن رمزهای عبور بدون استفاده از کتابخانه فوق در زبان‌های مختلف برنامه‌نویسی بررسی شده است.

۲ کتابخانه رمزنگاری sodium (libsodium)

Sodium یک کتابخانه نرم‌افزاری مدرن و قابل استفاده آسان برای رمزنگاری، رمزگشایی، امضای دیجیتال، تولید hash رمزهای عبور و مواردی از این قبیل می‌باشد. Sodium در واقع مشتق شده از NaCl[1] به همراه یک API مناسب و قابل توسعه است. NaCl یک کتابخانه جدید و سریع برای ارتباطات شبکه و مباحث مربوط به رمزنگاری می‌باشد. هدف sodium (و در واقع NaCl)، فراهم کردن تمامی فرآیندهای مورد نیاز برای ساخت ابزارهای رمزنگاری سطح بالا است.

Sodium از بسیاری از کامپایلرها و سیستم‌های‌عامل مانند ویندوز (به همراه MinGW یا Visual Studio، نسخه ۳۲ و ۶۴ بیتی)، IOS و آندروید پشتیبانی می‌کند.

در طراحی sodium امنیت به طور خاص مد نظر قرار گرفته است. علاوه بر آن طراحی به گونه‌ای انجام شده که دارای بالاترین سرعت و بیشترین کارایی باشد.

[1] Networking and Cryptography library

۲-۱ نحوه نصب

Sodium یک کتابخانه مشترک به همراه سرآیندهای مستقل از سخت‌افزار می‌باشد و بنابراین به راحتی می‌توان از آن در پروژه‌های گوناگون استفاده کرد.

بسته مربوط به libsodium را می‌توان از آدرس زیر دانلود نمود:

https://download.libsodium.org/libsodium/releases/

در مسیری که بسته مورد نظر دانلود شده است، با استفاده از دستورات زیر مراحل نصب دنبال می‌شود:

```
$ ./configure
$ make && make check
# make install
```

۲-۱-۱ Cross-compiling

Sodium به طور کامل از Cross-compiling پشتیبانی می‌کند. در زیر مثالی از Cross-compiling با استفاده از ابزار GNU برای پردازنده‌های تعبیه شده ARM آورده شده است:

```
$ export PATH=/path/to/gcc-arm-none-eabi/bin:$PATH
$ export LDFLAGS='--specs=nosys.specs'
$ export CFLAGS='-Os'
$ ./configure --host=arm-none-eabi --prefix=/install/path
$ make install
```

توجه شود که '--specs=nosys.specs' فقط برای ARM مورد نیاز است.

۲-۱-۲ کامپایل کردن با استفاده از CompCert

می‌توان توزیع‌ها را با استفاده از کامپایلر CompCert کامپایل کرد. هنگامی که از CompCert استفاده می‌شود، امکان تشخیص اشتباه در اسکریپت مربوط به تنظیمات big endian و little endian وجود دارد. برای رفع این مشکل می‌توان به طور دستی تنظیمات مربوط به LITTLE_ENDIAN را انجام داد.

در بخش زیر نمونه کد مربوط به کامپایل کردن sodium توسط کامپایلر CompCert بر روی یک سیستم little endian آورده شده است:

```
env CC=ccomp CPPFLAGS=-DLITTLE_ENDIAN \
   CFLAGS="-O2 -fstruct-passing" ./configure \
   --disable-shared --enable-static && \
make check && make install
```

توصیه می‌شود که برای نصب sodium از نسخه tarball به‌جای استفاده از نسخه موجود در مخزن git استفاده شود. نسخه tarball به پیش‌نیازهایی برای نصب مانند libtool یا autotool احتیاج ندارد.

۲-۱-۳ نحوه بررسی صحت فایل

فایل‌های توزیع دانلود شده را می‌توان توسط Minisign و کلید ED25519 زیر بررسی کرد:

RWQf6LRCGA9i53mlYecO4IzT51TGPpvWucNSCh1CBM0QTaLn73Y7GFO3

روش دیگر استفاده از GnuPG و کلید RSA ای است که در پیوست ۱ آمده است.

۲-۲ استفاده از کتابخانه libsodium در دیگر زبان‌ها

از این کتابخانه می‌توان در زبان‌های مختلف برنامه‌نویسی استفاده نمود. در لیست زیر، زبان‌ها (به همراه کتابخانه‌هایی که از طریق آن‌ها می‌توان از sodium استفاده کرد) فهرست شده است.

.NET: libsodium-net
C++: sodiumpp
C++: tears
Clojure: caesium
Clojure: naclj
Common LISP: cl-sodium
D: Chloride
D: Shaker
D: Sodium

Delphi/FreePascal: Delphi/FreePascal
Dylan: libsodium-dylan
Elixir: Savory
Erlang: ENaCl
Erlang: Erlang-NaCl
Erlang: Erlang-Libsodium
Erlang: Salt
Fortran: Fortium
Go: GoSodium
Go: libsodium-go
Go: Natrium
Go: Sodium
Haskell: Saltine
HaXe: haxe_libsodium
Idris: Idris-Sodium
Java (Android): Libstodium
Java (Android): Robosodium
Java (Android): Libsodium-JNI
Java: Kalium
Java: sodium-jni
JavaScript (compiled to pure JavaScript): libsodium.js
JavaScript (libsodium.js wrapper): Natrium
JavaScript (libsodium.js wrapper for browsers): Natrium Browser
JavaScript (NodeJS): node-sodium
Julia: Sodium.jl
Lisp (CFFI): foreign-sodium
Lua: lua-sodium
MRuby: mruby-libsodium
Nim: Libsodium.nim
Nim: Sodium.nim
OCaml: ocaml-sodium
Objective-C: NAChloride
Objective-C: SodiumObjc
PHP: PHP-Sodium
PHP: libsodium-php
Perl: Crypt-Sodium
Perl: Crypt::Nacl::Sodium

Pharo/Squeak: Crypto-NaCl
Pony: Pony-Sodium
Python: LibNaCl
Python: PyNaCl
Python: PySodium
Q/KDB: Qsalt
R: Sodium
Racket: Natrium
Racket: part of CRESTaceans
Ruby: RbNaCl
Ruby: Sodium
Rust: Sodium Oxide
Rust: libsodium-sys
Swift: NaOH
Swift: Swift-Sodium

۳-۲ نحوه استفاده

```
#include <sodium.h>
int main(void)
{
   if (sodium_init() == -1) {
      return 1;
   }
   ...
}
```

در این مثال فقط از سرآیند sodium.h استفاده شده است. برای لینک کردن کتابخانه sodium باید از-lsodium استفاده کرد. همچنین پرچم‌های مناسب مربوط به کامپایل‌کردن و یا لینک‌کردن با استفاده از دستور pkg-config به‌دست می‌آیند.

```
CFLAGS=$(pkg-config --cflags libsodium)
LDFLAGS=$(pkg-config --libs libsodium)
```

کاربرانی که از Visual Studio استفاده می‌کنند، برای static linking باید تنظیمات زیر را لحاظ کنند:

SODIUM_STATIC=1
SODIUM_EXPORT=

این تنظیمات بر روی پلتفرم‌های دیگر نیاز نمی‌باشد.
تابع sodium_init() مربوط به مقداردهی اولیه و راه‌اندازی کتابخانه sodium می‌باشد و باید قبل از استفاده از هر تابعی دیگر از کتابخانه sodium فراخوانی شود. این تابع می‌تواند بیشتر از یک بار فراخوانی شود ولی نمی‌تواند توسط چندین thread به طور همزمان اجرا گردد. اگر در برنامه مورد استفاده، حالتی که گفته شد اتفاق می‌افتد باید از قفل‌های مناسب در اطراف فراخوانی تابع استفاده شود.

sodium_init() هیچگونه تخصیص حافظه‌ای را انجام نمی‌دهد. در هر حال، در سیستم‌های مبتنی بر Unix یک فایل توصیف‌کننده در مسیر /dev/urandom باز می‌شود و بنابراین دستگاه بعد از فراخوانی chroot() همچنان قابل دسترس باقی می‌ماند. ضمناً فراخوانی‌های متعدد تابع sodium_init() باعث باز شدن چندین فایل توصیف‌کننده نمی‌شود.

تابع sodium_init() اگر به طور موفقیت‌آمیز اجرا شود، عدد 0 و در غیر اینصورت عدد 1- را باز می‌گرداند. همچنین اگر تابع قبلاً فراخوانی شده باشد، عدد 1 را باز می‌گرداند.

۴-۲ Helpers

هنگامی که مقایسه شامل عبارات محرمانه مانند کلیدها و یا برچسب‌های مربوط به احراز اصالت می‌باشد، ضروری است که به طور زمان-ثابت از توابع مقایسه‌ای به منظور کاهش حمله side-channel استفاده شود. تابع sodium-memcmp() برای این منظور استفاده می‌شود.

تست‌های زمان-ثابت به منظور برابری:
int sodium_memcmp(const void * const b1_, const void * const b2_, size_t len);

در این تابع، اگر اندازه بایت‌هایی که توسط _b1 به آن‌ها اشاره می‌شود با اندازه بایت‌هایی که توسط _b2 به آن‌ها اشاره می‌شود برابر باشند، عدد 0 و در غیر اینصورت عدد 1- را باز می‌گرداند.

توجه شود که تابع sodium_memcmp() جایگزینی برای تابع memcmp() نمی‌باشد.

۱-۴-۲ Hexadecimal encoding/decoding

```
char *sodium_bin2hex(char * const hex, const size_t hex_maxlen,
                     const unsigned char * const bin, const size_t bin_len);
```

تابع sodium_bin2hex() بایت‌های bin_len که در محل bin ذخیره شده‌اند را به یک رشته هگزادسیمال تبدیل می‌کند. این رشته در محل hex ذخیره می‌شود و در آخر آن به عنوان اتمام رشته از یک بایت null (\0) استفاده می‌کند.

hex_maxlen حداکثر تعداد بایت‌هایی است که تابع می‌تواند در محل hex ذخیره کند. این مقدار باید حداقل به اندازه bin_len * 2 + 1 باشد.

این تابع در صورت عملکرد موفقیت‌آمیز، مقدار hex و در صورت بروز overflow، null را باز می‌گرداند. تابع مورد نظر در زمان‌های منظم و برای یک اندازه خاص ارزیابی می‌شود.

```
int sodium_hex2bin(unsigned char * const bin, const size_t bin_maxlen,
                   const char * const hex, const size_t hex_len,
                   const char * const ignore, size_t * const bin_len,
                   const char ** const hex_end);
```

تابع sodium_hex2bin() رشته هگزادسیمال hex را تجزیه و آن را به دنباله‌ای از بایت‌ها تبدیل می‌کند.

رشته hex برای خاتمه آن، از کاراکتر null استفاده نمی‌کند و تعداد کاراکترهایی که باید تجزیه شوند توسط پارامتر hex_len تعیین می‌شود.

ignore یک رشته از کاراکترهایی است که می‌توان از آن‌ها عبور کرد. برای مثال، رشته " :" به کاراکتر : و 'فاصله' این اجازه را می‌دهد که در هر مکانی از رشته هگزادسیمال ظاهر شود. تنها این کاراکترها نادیده گرفته می‌شوند. به عنوان نتیجه "69:FC", "69 FC", "69 : FC" و "69FC" همگی ورودی‌های معتبری می‌باشند که یک خروجی یکسان را تولید می‌کنند.

برای اینکه هیچ کاراکتر غیر هگزادسیمال پذیرفته نشود، می‌توان ignore را با null مقداردهی کرد. همچنین bin_maxlen تعیین‌کننده تعداد حداکثر بایت‌هایی است که در bin می‌توان ذخیره کرد.

عملیات بررسی رشته مورد نظر زمانی متوقف می‌شود که یا یک کاراکتر غیر هگزادسیمال و یا کاراکترهایی که ignore نیستند، یافت شود و یا زمانی که تعداد حداکثر بایت‌هایی که توسط bin_maxlen مشخص شده‌اند، در bin نوشته شود.

اگر تعداد بایت‌هایی که قرار است پس از تجزیه رشته مورد نظر ذخیره شوند، بیشتر از حداکثرتعداد مجازی باشد که توسط bin_maxlen مشخص شده است، تابع مقدار 1- را باز می‌گرداند. اگر تابع به طور صحیح اجرا شود، مقدار 0 بازگردانده می‌شود و اگر hex_end مقدار null نداشته باشد، با یک مقدار پر می‌شود. این مقدار در واقع یک اشاره‌گر به کاراکتری است که بعد از آخرین کاراکتر تجزیه شده قرار دارد.

این تابع در زمان‌های ثابت و با توجه به طول و فرمت مشخص، اجرا می شود.

۲-۴-۲ افزایش عددهای بزرگ

void sodium_increment(unsigned char *n, const size_t nlen);

تابع sodium_increment() یک اشاره‌گر دریافت می‌کند که به عددی بدون علامت و بزرگ اشاره می‌کند و آن عدد را افزایش می‌دهد.

این تابع در زمان‌های مشخص برای طول‌های معین اجرا می‌شود و اینگونه فرض می‌شود که عدد مورد نظر به فرمت little-endian باشد.

از این تابع می‌توان برای افزایش مقدار nonce ها در زمان‌های ثابت استفاده کرد.

۲-۴-۳ اضافه‌کردن اعداد بزرگ

void sodium_add(unsigned char *a, const unsigned char *b, const size_t len);

تابع sodium_add() دو اشاره‌گر به عنوان ورودی دریافت می‌کند. این دو اشاره‌گر به اعداد بدون علامتی اشاره می‌کنند که ساختار little-endian دارند. در دستور بالا هر دو اشاره‌گر a و b دارای اندازه len می‌باشند.

این تابع مقدار (a + b) mod 2^(8*len) را در زمان‌های ثابت و با توجه به طول مشخص محاسبه می‌کند و نتیجه را بر روی a بازنویسی می‌کند.

۲-۴-۴ مقایسه اعداد بزرگ

int sodium_compare(const void * const b1_, const void * const b2_, size_t len);

با توجه به مقدار b1_ و b2_ که دو عدد با اندازه len و به فرمت little-endian می‌باشند، این تابع مقادیر زیر را باز می‌گرداند:

- 1- اگر b1_ کوچکتر از b2_ باشد،
- 0 اگر b1_ برابر b2_ باشد،

- 1 اگر _b1 بزرگتر از _b2 باشد.

این مقایسه در زمان‌های ثابت و برای طول‌های مشخص انجام می‌شود. این تابع می‌تواند به همراه nonce ها و به منظور جلوگیری از حمله replay استفاده شود.

۲-۴-۵ بررسی به منظور صفربودن

int sodium_is_zero(const unsigned char *n, const size_t nlen);

این تابع در صورتی که تمام بایت‌های nlen که توسط n به آن‌ها اشاره می‌شود صفر باشد، مقدار 1 را باز می‌گرداند. اگر بایت‌های مورد بررسی مقداری غیر صفر داشته باشند، عدد 0 باز گردانده می‌شود.

۲-۵ تخصیص حافظه امن

۲-۵-۱ پاک‌کردن حافظه

void sodium_memzero(void * const pnt, const size_t len);

پس از استفاده از برنامه مورد نظر، باید اطلاعات حساس مجدداً بازنویسی شوند، ولی memset() و کدهای دست‌نوشته شده به دلیل بهینه‌سازی کامپایلر و یا لینکر می‌توانند از این قاعده مستثنی شوند.

تابع sodium_memzero() به طور مؤثر تعداد len بایت از نقطه شروع pnt را صفر می‌کند (حتی اگر بهینه‌سازی بر روی کد اعمال شده باشد).

۲-۵-۲ قفل حافظه

int sodium_mlock(void * const addr, const size_t len);

تابع sodium_mlock() تعداد حداقل len بایت از نقطه شروع addr را قفل می‌کند. این عمل می‌تواند از جابه‌جایی اطلاعات حساس به hard disk جلوگیری کند. این جابه‌جایی به این دلیل می‌تواند انجام شود که در تنظیمات سیستم، swap فعال شده باشد.

توصیه می‌شود که در تنظیمات سیستم‌هایی که اطلاعات حساس بر روی آن‌ها پردازش می‌شود، یا swap به طور کامل غیر فعال شود و یا از یک پارتیشن swap رمزشده استفاده گردد.

به دلیل مشابه در سیستم‌های مبتنی بر یونیکس، هنگامی که کدهای رمزنگاری خارج از محیط توسعه نرم‌افزار اجرا می‌شوند، باید تنظیمات مربوط به dump گرفتن از هسته غیر فعال شود. این عمل را می‌توان از طریق استفاده از دستور ulimit که در shell به صورت پیش‌فرض وجود دارد و یا با استفاده از دستور setrlimit(RLIMIT_CORE, &(struct rlimit) {0, 0}) انجام داد. در سیستم‌عامل‌هایی که این ویژگی در آن‌ها پیاده‌سازی شده است، تنظیمات مربوط به dump گرفتن از crash های مربوط به کرنل نیز باید غیر فعال گردند.

sodium_mlock() ترکیبی از mlock() و VirtualLock() می‌باشد. توجه شود که بسیاری از سیستم‌ها بر روی میزان حافظه‌ای که توسط یک پردازش می‌تواند اشغال شود، محدودیت می‌گذارند. باید دقت شود که این محدودیت‌ها در صورت نیاز افزایش پیدا کنند. تابع sodium_lock() زمانی که به دلیل این محدودیت‌ها به طور موفقیت‌آمیز نتواند فعالیت خود را انجام دهد، عدد 1- را باز می‌گرداند.

int sodium_munlock(void * const addr, const size_t len);

تابع sodium_munlock() باید زمانی فراخوانی شود که از عدم استفاده از حافظه‌ای که قفل شده است، مطمئن بود. این تابع تعداد len بایت از نقطه شروع addr را قبل از آنکه این صفحه را مجدداً به عنوان قابل swap شدن علامت بزند، صفر می‌کند. بنابراین نیازی به فراخوانی تابع sodium_memzero() قبل از sodium_munlock() نمی‌باشد.

۲-۵-۳ تخصیص پشته محافظت‌شده

sodium توابعی برای تخصیص پشته به منظور ذخیره اطلاعات حساس فراهم می‌کند. این توابع در واقع توابع تخصیص همه منظوره و کلی نیستند. به طور خاص، این توابع کندتر از ()malloc و مشابه آن عمل می‌کنند و به 3 و یا 4 صفحه اضافی از حافظه مجازی نیاز دارند.

()sodium_init باید قبل از استفاده از توابع مربوط به تخصیص حافظه محافظت‌شده فراخوانی شود.

void *sodium_malloc(size_t size);

تابع ()sodium_malloc به عنوان خروجی، یک اشاره‌گر باز می‌گرداند که به بایت‌های به هم پیوسته (دقیقاً به اندازه size از حافظه که قابل دسترس می‌باشند) اشاره می‌کند.

ناحیه تخصیص داده شده در انتهای صفحه و بلافاصله پس از صفحه محافظت‌شده قرار می‌گیرد. به عنوان نتیجه، دسترسی به فضای بعد از ناحیه محافظت‌شده، بلافاصله باعث خارج شدن از برنامه می‌شود.

یک نشانگر دقیقاً قبل از اشاره‌گر خروجی قرار داده می‌شود. تغییر این نشانگر زمانی که سعی در آزاد کردن ناحیه تخصیص یافته با استفاده از تابع ()sodium_free وجود داشته باشد، شناخته شده و باعث خاتمه فوری برنامه می‌شود.

یک صفحه حفاظتی اضافی نیز قبل از این نشانگر قرارداده می‌شود تا احتمال دسترسی به اطلاعات حساس هنگامی که بعد از یک ناحیه نامربوط خوانده می‌شود، کاهش یابد.

ناحیه تخصیص یافته با بایت‌های 0xd0 پر می‌شود. این کار به دلیل کمک به فهمیدن خطاهای وابسته به زمان شروع برنامه و مقداردهی اولیه به متغیرها انجام می‌شود.

علاوه بر این، sodium_mlock() در ناحیه مورد نظر به منظور جلوگیری از swap شدن این ناحیه بر روی هارد دیسک فراخوانی می‌شود. بر روی سیستم‌عامل‌هایی که از MAP_NOCORE یا MADV_DONTDUMP پشتیبانی می‌کنند، حافظه تخصیص داده شده با این روش جزء نمونه‌های گرفته شده از هسته (core dump) نیستند.

اگر اندازه تخصیص یافته شده مضربی از مقدار مورد نیاز نباشد، آدرس بازگردانده شده به صورت مورد انتظار نخواهد بود. به همین دلیل، sodium_malloc() نباید به همراه ساختارهای بسته‌بندی شده و یا با طول متغیر استفاده شود، مگر اینکه اندازه‌ای که به تابع sodium_malloc() داده می‌شود برای بازگرداندن آدرس مورد انتظار، گرد شود.

تمام ساختارهایی که توسط libsodium استفاده می‌شوند به طور امن می‌توانند توسط تابع sodium_mallloc() تخصیص داده شوند. تنها حالتی که نیاز به دقت بیشتری دارد، حالت crypto_generichash_state می‌باشد. در این حالت باید اندازه فضا را به مضربی از 64 بایت تغییر داد.

```
void *sodium_allocarray(size_t count, size_t size);
```

تابع sodium_allocarray() یک اشاره‌گر به عنوان خروجی باز می‌گرداند که به object هایی (count) که دارای اندازه size بایت از حافظه هستند و قابل دسترس می‌باشند، اشاره می‌کند.

```
void sodium_free(void *ptr);
```

تابع sodium_free() حافظه‌ای را که توسط تابع sodium_malloc() یا sodium_allocarray() تخصیص داده شده است، از حالت قفل خارج و آن را آزاد می‌کند.

پیش از این، نشانگر گفته شده به منظور پاریز احتمالی بافر (buffer underflow) و خاتمه پردازش مربوطه در صورت نیاز، بررسی می‌شود.

همچنین sodium_free() قبل از آزاد سازی حافظه، ناحیه مورد نظر را با 0 پر می‌کند.

این تابع حتی اگر ناحیه مورد نظر قبلاً با استفاده از sodium_mprotect_readonly() محافظت شده باشد، می‌تواند فراخوانی شود. پس از آن نحوه حفاظت به صورت خودکار تغییر می‌کند.

ptr می‌تواند NULL باشد، در این حالت هیچ عملیاتی انجام نمی‌شود.

int sodium_mprotect_noaccess(void *ptr);

تابع sodium_mprotect_noaccess() ناحیه تخصیص‌یافته با استفاده از توابع sodium_malloc() و یا sodium_allocarray() را غیرقابل دسترس می‌کند. در نتیجه این ناحیه نمی‌تواند خوانده و یا روی آن چیزی نوشته شود، ولی اطلاعات قبلی محفوظ باقی می‌مانند.

این تابع می‌تواند اطلاعات محرمانه را غیر قابل دسترس کند، مگر اینکه برای انجام عملیاتی خاص واقعاً به آن‌ها نیاز باشد.

int sodium_mprotect_readonly(void *ptr);

تابع sodium_mprotect_readonly() یک ناحیه که توسط تابع sodium_malloc() و یا sodium_allocarray() تخصیص یافته است را در حالت فقط خواندنی (read-only) علامت می‌گذارد. تلاش برای تغییر اطلاعات باعث خاتمه پردازش می‌شود.

int sodium_mprotect_readwrite(void *ptr);

تابع sodium_mprotect_readwrite() یک ناحیه که توسط تابع sodium_malloc() و یا sodium_allocarray() تخصیص یافته است را پس از این‌که توسط توابع sodium_mprotect_readonly() و یا sodium_mprotect_noaccess() محافظت شده است، در حالت‌های قابل خواندن و نوشتن (readable & writable) علامت‌گذاری می‌کند.

۲-۶ تولید اعداد تصادفی

کتابخانه sodium مجموعه‌ای از توابع به منظور تولید اطلاعات غیرقابل پیش‌بینی و مناسب برای ساخت کلیدهای رمزنگاری را فراهم می‌کند. در زیر این توابع برای هرکدام از سیستم‌های عامل مرسوم ذکر شده‌اند:

- در سیستم‌های ویندوزی از تابع RtlGenRandom() استفاده شده است.
- در سیستم‌عامل OpenBSD و Bitrig از تابع arc4random استفاده شده است.
- در کرنل‌های اخیر لینوکس از یک system call به نام getrandom استفاده شده است.
- در دیگر سیستم‌های Unix از /dev/urandom استفاده شده است.
- اگر هیچکدام از این موارد به طور امن نتوانند استفاده شوند، یک پیاده‌سازی سفارشی می‌تواند به کار آید.

۲-۶-۱ نحوه استفاده

```
uint32_t randombytes_random(void);
```

تابع randombytes_random() یک مقدار غیرقابل پیش‌بینی بین 0 و 0xffffffff باز می‌گرداند.

```
uint32_t randombytes_uniform(const uint32_t upper_bound);
```

تابع randombytes_uniform() یک مقدار تصادفی بین 0 و upper_bound باز می‌گرداند. بر خلاف تابع randombytes_random()، تابع randombytes_uniform() توزیع یکنواخت بهتری از خروجی‌های ممکن را در اختیار می‌گذارد.

```
void randombytes_buf(void * const buf, const size_t size);
```

تابع randombytes_buf() فضای حافظه به اندازه size بایت از نقطه شروع buf را با یک دنباله از بایت‌های غیرقابل پیش‌بینی پر می‌کند.

int randombytes_close(void);

این تابع فضای کلی که توسط مولد اعداد شبه تصادفی استفاده شده بود را آزاد می‌کند. به خصوص هنگامی که تجهیز /dev/urandom استفاده شده باشد، باعث بسته شدن توصیف‌گر (descriptor) می‌شود. واضح است که فراخوانی این تابع هرگز احتیاج نخواهد بود.

void randombytes_stir(void);

تابع randombytes_stir() در صورت پشتیبانی، عدد پایه استفاده شده در مولد اعداد شبه تصادفی را تغییر می‌دهد. فراخوانی این تابع به همراه مولد پیش‌فرض حتی پس از فراخونی تابع fork() نیاز نمی‌باشد. البته مگر اینکه فایل توصیف‌گر برای /dev/random با استفاده از randombytes_close() بسته شده باشد.

اگر پیاده‌سازی پیش‌فرض استفاده نشده باشد، باید randombytes_stir() پس از فراخوانی تابع fork() توسط زیرتابع مربوطه فراخوانی شود.

توجه: اگر توابع این قسمت در برنامه‌ای درون یک ماشین مجازی استفاده شده باشد و از ماشین مجازی snapshot گرفته و دوباره بازیابی شود، ممکن است توابع بالا خروجی یکسان تولید کنند.

۲-۷ رمزنگاری کلید خصوصی

۲-۷-۱ رمزنگاری عملیات مربوط به احراز اصالت کلید

مثال:

```
#define MESSAGE ((const unsigned char *) "test")
#define MESSAGE_LEN 4
#define CIPHERTEXT_LEN (crypto_secretbox_MACBYTES + MESSAGE_LEN)

unsigned char nonce[crypto_secretbox_NONCEBYTES];
```

```
unsigned char key[crypto_secretbox_KEYBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];

randombytes_buf(nonce, sizeof nonce);
randombytes_buf(key, sizeof key);
crypto_secretbox_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
if (crypto_secretbox_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce, key) != 0) {
   /* message forged! */
}
```

هدف از انجام این عملیات عبارت است از:

- رمزکردن پیام توسط یک کلید و یک nonce برای حفظ محرمانگی آن
- محاسبه برچسب احراز اصالت (این برچسب برای حصول اطمینان از عدم دستکاری پیام با استفاده از رمزگشایی آن می‌باشد).

از آن‌جایی‌که از یک کلید یکسان برای عملیات رمزنگاری و امضاء و همچنین تصدیق و رمزگشایی پیام استفاده می‌شود، به همین دلیل حفظ محرمانگی کلید یک امر ضروری می‌باشد.

مقدار nonce فاش می‌باشد و نیازی به محرمانه بودن آن نیست، ولی به هیچ عنوان نباید برای یک کلید یکسان از nonce قبلی آن مجدداً استفاده کرد. راحت‌ترین روش برای تولید nonce ، استفاده از تابع randombytes_buf() می‌باشد.

حالت ترکیبی: در این حالت، برچسب احراز اصالت و پیام رمزشده باهم ذخیره می‌شوند:

```
int crypto_secretbox_easy(unsigned char *c, const unsigned char *m,
                          unsigned long long mlen, const unsigned char *n,
                          const unsigned char *k);
```

تابع crypto_secretbox_easy() پیام m به طول meln بایت را با کلید k و نانس n رمز می‌کند.

k باید بایت‌های crypto_secretbox_KEYBYTES و n باید بایت‌های crypto_secretbox_NONCEBYTES باشد. همچنین c باید حداقل دارای طول mlen + crypto_secretbox_MACBYTES بایت باشد.

این تابع برچسب احراز اصالت دارای طول crypto_secretbox_MACBYTES بایت می‌باشد و بلافاصله پس از آن پیام رمزشده که دارای طول mlen و مشابه با طول پیام فاش می‌باشد را در c می‌نویسد.

```
int crypto_secretbox_open_easy(unsigned char *m, const unsigned char *c,
                               unsigned long long clen, const unsigned char *n,
                               const unsigned char *k);
```

تابع crypto_secretbox_open_easy() متن رمزشده‌ای که توسط تابع crypto_secretbox_easy() تولید شده است را تصدیق و رمزگشایی می‌کند.

c در واقع یک اشاره‌گر به ترکیبی از برچسب احراز اصالت و متن رمزشده می‌باشد که توسط تابع crypto_secretbox_easy() تولید شده است. طول این ترکیب برچسب احراز اصالت و متن رمزشده clen می‌باشد.

به عبارت دیگر، clen تعداد بایت‌های نوشته شده توسط تابع crypto_secretbox_easy() است که برابر با تعداد بایت‌های crypto_secretbox_MACBYTES به‌علاوه طول پیام می‌باشد.

نانس n و کلید k باید برای رمزنگاری و احراز اصالت پیام مطابقت داشته باشند.

اگر نتیجه تصدیق منفی بود، تابع عدد 1- و در غیر این صورت عدد 0 را به عنوان خروجی باز می‌گرداند. در صورت موفقیت‌آمیز بودن این بررسی، پیام رمزگشایی شده در m ذخیره می‌شود.

حالت منفصل یا مجزا: بعضی از برنامه‌ها نیازمند حالتی می‌باشند که برچسب احراز اصالت و پیام رمزشده در مکان‌های مختلف ذخیره شوند. برای این حالت خاص، نوع مجزای تابع بالا قابل استفاده می‌باشد.

```
int crypto_secretbox_detached(unsigned char *c, unsigned char *mac,
                              const unsigned char *m,
                              unsigned long long mlen,
                              const unsigned char *n,
                              const unsigned char *k);
```

این تابع پیام m به طول mlen بایت را با کلید k و نانس n رمز و پیام رمزشده را در c ذخیره می‌کند. توجه شود که دقیقاً mlen بایت در c ذخیره می‌شود. همچنین برچسب احراز اصالت که اندازه آن crypto_secretbox_MACBYTES بایت است در mac ذخیره می‌شود.

```
int crypto_secretbox_open_detached(unsigned char *m,
                                   const unsigned char *c,
                                   const unsigned char *mac,
                                   unsigned long long clen,
                                   const unsigned char *n,
                                   const unsigned char *k);
```

تابع crypto_secretbox_open_detached() پیام رمزشده c با طول clen را ابتدا بررسی و تصدیق و سپس آن را رمزگشایی می‌کند. clen حاوی برچسب احراز اصالت نمی‌باشد، بنابراین طول پیام رمزشده با طول پیام فاش یکسان است.

پیام فاش پس از بررسی و تصدیق اینکه mac یک برچسب احراز اصالت معتبر برای این پیام رمزشده می‌باشد (با در نظر داشتن نانس n و کلید k)، در m ذخیره می‌شود.

این تابع در صورت موفقیت‌آمیز بودن نتیجه بررسی و تصدیق، عدد 0 و در غیر این صورت عدد 1- را باز می‌گرداند.

مقادیر ثابت:

- crypto_secretbox_KEYBYTES
- crypto_secretbox_MACBYTES
- crypto_secretbox_NONCEBYTES

جزئیات الگوریتم:

- Encryption: XSalsa20 stream cipher
- Authentication: Poly1305 MAC

توجه: نسخه اصلی crypto_secretbox که یک API مربوط به NaCl است نیز پشتیبانی می‌شود.

تابع crypto_secretbox() یک اشاره‌گر به فضای ۳۲ بایتی قبل از پیام دریافت می‌کند و متن رمزشده را در مکانی معادل ۱۶ بایت پس از اشاره‌گر مقصد ذخیره می‌کند، همچنین ۱۶ بایت ابتدایی نیز با 0 مجدداً بازنویسی می‌شود.

تابع crypto_secretbox_open() یک اشاره‌گر به فضای ۱۶ بایتی قبل از متن رمزشده دریافت می‌کند و پیام را در مکانی معادل ۳۲ بایت پس از اشاره‌گر مقصد ذخیره می‌کند. همچنین ۳۲ بایت ابتدایی با 0 مجدداً بازنویسی می‌شود.

۲-۷-۲ احراز اصالت کلید خصوصی

مثال:

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char key[crypto_auth_KEYBYTES];
unsigned char mac[crypto_auth_BYTES];

randombytes_buf(key, sizeof key);
crypto_auth(mac, MESSAGE, MESSAGE_LEN, key);
```

```
if (crypto_auth_verify(mac, MESSAGE, MESSAGE_LEN, key) != 0) {
   /* message forged! */
}
```

در این عملیات، یک برچسب احراز اصالت برای پیام و کلید خصوصی محاسبه می‌شود و همچنین روشی برای بررسی اعتبار برچسب احراز اصالت برای یک پیام و کلید آن فراهم می‌شود.

این تابع برچسب احراز اصالت را به طور قطعی تعیین می‌کند: هر زوج "پیام،کلید" مشابه همیشه خروجی یکسانی تولید می‌کند.

به هر حال حتی اگر پیام عمومی باشد، دانستن کلید برای محاسبه برچسب معتبر مورد نیاز می‌باشد. بنابراین باید کلید محرمانه باقی بماند ولی برچسب می‌تواند فاش باشد.

یک سناریوی نمونه می‌تواند به شرح زیر باشد:

- طرف A یک پیام آماده می‌کند، برچسب احراز اصالت را به آن اضافه و آن را برای طرف B ارسال می‌کند.
- A پیام را ذخیره نمی‌کند.
- پس از آن B پیام و برچسب احراز اصالت را به طرف A ارسال می‌کند.
- A با استفاده از برچسب احراز اصالت بررسی می‌کند که آیا او خودش قبلاً پیام را ساخته است یا خیر.

این فرآیند پیام را رمز نمی‌کند، بلکه تنها برچسب احراز اصالت را محاسبه و بررسی و تصدیق می‌کند.

نحوه استفاده:

```
int crypto_auth(unsigned char *out, const unsigned char *in,
        unsigned long long inlen, const unsigned char *k);
```

تابع crypto_auth() برای پیام in به طول inlen بایت و کلید k یک برچسب تولید می‌کند. k باید crypto_auth_KEYBYTES بایت باشد. این تابع، برچسب محاسبه شده را در out ذخیره می‌کند. همچنین برچسب تولید شده دارای طول crypto_auth_BYTES بایت می‌باشد.

```
int crypto_auth_verify(const unsigned char *h, const unsigned char *in,
                       unsigned long long inlen, const unsigned char *k);
```

تابع crypto_auth_verify() بررسی می‌کند که آیا برچسب ذخیره شده در h یک برچسب معتبر برای پیام in به طول inlen بایت و کلید k است یا خیر. اگر نتیجه موفقیت‌آمیز باشد، تابع عدد 0 را باز می‌گرداند و در غیر این صورت عدد 1- به عنوان خروجی بازگردانده می‌شود.

ثابت‌ها:

- crypto_auth_BYTES
- crypto_auth_KEYBYTES

جزئیات الگوریتم:

- HMAC-SHA512256

۲-۷-۳ رمزنگاری احراز اصالت شده همراه با اطلاعات اضافی

این فرآیند شامل مراحل زیر می‌باشد:

- رمزنگاری پیام با یک کلید و نانس برای حفظ محرمانگی آن
- محاسبه برچسب احراز اصالت: این امر برای حصول اطمینان از عدم دستکاری پیام می‌باشد.

یک نمونه از اطلاعات اضافی مورد نظر می‌تواند داده‌های مربوط به یک پروتکل خاص درباره پیام باشد، مانند طول و نحوه کدگذاری آن.

ساخت‌های پشتیبانی‌شده: Libsodium از دو ساخت مشهور AES256-GCM و ChaCha20-Poly1305 پشتیبانی می‌کند:

- AES256-GCM : پیاده‌سازی فعلی از این حالت، سخت‌افزاری و نیازمند اضافه‌کردن سخت‌افزار Intel SSSE3 می‌باشد. در حال حاضر روشی به غیر از روش سخت‌افزاری برای پیاده‌سازی این حالت وجود ندارد. اگر ویژگی portability مد نظر نباشد، AES256-GCM سریعترین گزینه می‌باشد.

- ChaCha20-Poly1305 : در حالی که AES بر روی سخت‌افزار اختصاصی بسیار سریع می‌باشد ولی کارایی و بازدهی آن بر روی پلتفرم‌هایی که فاقد سخت‌افزار مورد نظر باشند، به طور قابل توجهی کاهش می‌یابد. مشکل دیگر، آسیب‌پذیری بسیاری از پیاده‌سازی‌های نرم‌افزاری AES در برابر حمله cache-collision timing می‌باشد. ChaCha20 به میزان قابل توجهی از پیاده‌سازی نرم‌افزاری AES سریع‌تر می‌باشد. این سرعت در بعضی موارد به میزان سه برابر نیز می‌رسد. همچنین ChaCha20 نسبت به timing attack آسیب‌پذیر نمی‌باشد. Poly1305 نیز یک پیام کد احراز اصالت می‌باشد. ترکیب متن رمزشده توسط ChaCha20 و تأیید Poly1305 در سال ۲۰۱۴ به عنوان یک جایگزین سریعتر برای روش Salsa20-Poly1305 پیشنهاد داده شده است. ChaCha20-Poly1305 در بیشتر سیستم‌عامل‌ها، مرورگرهای وب و کتابخانه‌های رمزنگاری پیاده‌سازی شده است. این روش در ماه می سال ۲۰۱۵ به عنوان استاندارد رسمی IETF تأیید شده است.

پیاده‌سازی ChaCha20-Poly1305 در Libsodium در تمام ساختارهای پشتیبانی شده قابل انجام می‌باشد و برای بیشتر برنامه‌ها استفاده از این روش توصیه شده است.

۲-۷-۳-۱ رمزنگاری احراز اصالت شده همراه با اطلاعات اضافی با استفاده از ChaCha20-Poly1305

این فرآیند شامل مراحل زیر می‌باشد:

- رمزنگاری پیام با یک کلید و نانس برای حفظ محرمانگی آن
- محاسبه برچسب احراز اصالت: این امر برای حصول اطمینان از عدم دستکاری پیام می‌باشد.

یک نمونه از اطلاعات اضافی مورد نظر می‌تواند داده‌های مربوط به یک پروتکل خاص درباره پیام مانند طول و نحوه کدگذاری آن باشد.

Libsodium دو نسخه از ساختار ChaCha20-Poly1305 را پیاده‌سازی می‌کند:

- نسخه اصلی می‌تواند به طور امن و بدون هیچ محدودیت عملی در مورد اندازه پیام، تعداد ۲۶۴ پیام را توسط یک کلید رمز کند. اندازه پیام می‌تواند حداکثر ۲۷۰ بایت باشد.
- نسخه IETF کمی کندتر می‌باشد. این نسخه می‌تواند به طور امن و به صورت عملی تعداد نامحدودی پیام (۲۹۶) را رمز کند. البته اندازه هر کدام از پیام‌ها نمی‌توانند از یک ترا بایت تجاوز کنند.

هر دو روش با دیگر کتابخانه‌های رمزنگاری نیز سازگار می‌باشند. ضمناً ویژگی‌های امنیتی یکسانی را به اشتراک می‌گذارند و از طریق API های مشابه قابل دسترسی هستند.

مجموعه توابع crypto_aead_chacha20poly1305_* () ، ساختار اصلی را پیاده‌سازی کرده در حالی که توابع crypto_aead_chacha20poly1305_ietf_* () نسخه IETF را پیاده‌سازی می‌کنند. ثابت‌ها به غیر از اندازه نانس، بدون تغییر باقی می‌مانند.

۲-۳-۷-۲ نسخه اصلی ChaCha20-Poly1305

مثال (حالت ترکیبی):

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define ADDITIONAL_DATA (const unsigned char *) "123456"
#define ADDITIONAL_DATA_LEN 6

unsigned char nonce[crypto_aead_chacha20poly1305_NPUBBYTES];
unsigned char key[crypto_aead_chacha20poly1305_KEYBYTES];
unsigned char ciphertext[MESSAGE_LEN + crypto_aead_chacha20poly1305_ABYTES];
unsigned long long ciphertext_len;
```

```
randombytes_buf(key, sizeof key);
randombytes_buf(nonce, sizeof nonce);

crypto_aead_chacha20poly1305_encrypt(ciphertext, &ciphertext_len,
                                     MESSAGE, MESSAGE_LEN,
                                     ADDITIONAL_DATA, ADDITIONAL_DATA_LEN,
                                     NULL, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
unsigned long long decrypted_len;
if (crypto_aead_chacha20poly1305_decrypt(decrypted, &decrypted_len,
                                         NULL,
                                         ciphertext, ciphertext_len,
                                         ADDITIONAL_DATA,
                                         ADDITIONAL_DATA_LEN,
                                         nonce, key) != 0) {
  /* message forged! */
}
```

در حالت ترکیبی، برچسب احراز اصالت و پیام رمزشده با یکدیگر و در یک مکان ذخیره می‌شوند.

```
int crypto_aead_chacha20poly1305_encrypt(unsigned char *c,
                                         unsigned long long *clen,
                                         const unsigned char *m,
                                         unsigned long long mlen,
                                         const unsigned char *ad,
                                         unsigned long long adlen,
                                         const unsigned char *nsec,
                                         const unsigned char *npub,
                                         const unsigned char *k);
```

تابع crypto_aead_chacha20poly1305_encrypt() پیام m به طول mlen بایت را با استفاده از کلید خصوصی k (crypto_aead_chacha20poly1305_KEYBYTES بایت) و نانس npub

(crypto_aead_chacha20poly1305_NPUBBYTES بایت) رمز می‌کند. پیام رمز شده و همچنین برچسب احراز اصالت که هم پیام محرمانه m و هم داده‌های غیرمحرمانه ad به طول adlen را احراز اصالت می‌کند، در c ذخیره می‌شوند.

ad می‌تواند یک اشاره‌گر NULL باشد و اگر اطلاعات اضافی مورد نیاز نباشند، اندازه adlen نیز می‌تواند برابر 0 باشد.

حداکثر تعداد بایت‌هایی که در c ذخیره می‌شوند و همینطور تعداد بایت‌های واقعی که در clen ذخیره می‌شوند (مگر اینکه lcen یک اشاره‌گر NULL باشد)، برابر با mlen + crypto_aead_chacha20poly1305_ABYTES می‌باشد.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

نانس npub که به صورت عمومی نیز در دسترس است، نباید هیچ موقع در ساخت یک کلید یکسان مجدداً استفاده شود. روش پیشنهادی برای تولید آن استفاده از تابع randombytes_buf() برای اولین پیام و اضافه کردن آن برای هرکدام از پیام‌های زیرمجموعه‌ای است که از یک کلید یکسان استفاده می‌کنند.

```
int crypto_aead_chacha20poly1305_decrypt(unsigned char *m,
                                         unsigned long long *mlen,
                                         unsigned char *nsec,
                                         const unsigned char *c,
                                         unsigned long long clen,
                                         const unsigned char *ad,
                                         unsigned long long adlen,
                                         const unsigned char *npub,
                                         const unsigned char *k);
```

تابع crypto_aead_chacha20poly1305_decrypt() تصدیق می‌کند که آیا متن رمزشده c (که توسط تابع crypto_aead_chacha20poly1305_encrypt() تولید شده است) شامل یک برچسب معتبر می‌باشد که از کلید خصوصی k، نانس npub و اطلاعات اضافی ad به طول adlen بایت استفاده کرده است.

ad می‌تواند یک اشاره‌گر NULL باشد و اگر اطلاعات اضافی مورد نیاز نباشد، اندازه adlen نیز می‌تواند برابر 0 باشد.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

اگر نتیجه بررسی موفقیت‌آمیز نباشد، تابع عدد 1- را به عنوان خروجی باز می‌گرداند. در غیر این صورت، عدد 0 بازگردانده می‌شود و پیام رمزگشایی شده در m ذخیره می‌شود. همچنین تعداد بایت‌های واقعی پیام ذخیره شده در m، در mlen ذخیره می‌شود. البته این در حالی است که meln یک اشاره‌گر NULL نباشد.

حداکثر بایت‌هایی که در m می‌تواند ذخیره شود برابر با clen - crypto_aead_chacha20poly1305_ABYTES است.

حالت مجزا:

بعضی از برنامه‌ها ممکن است که نیاز به ذخیره پیام رمزشده و برچسب احراز اصالت در دو مکان مختلف داشته باشند. برای این مورد، نوع مجزای تابع گفته شده در قسمت قبل وجود دارد.

```
int crypto_aead_chacha20poly1305_encrypt_detached(unsigned char *c,
                                                  unsigned char *mac,
                                                  unsigned long long *maclen_p,
                                                  const unsigned char *m,
                                                  unsigned long long mlen,
                                                  const unsigned char *ad,
                                                  unsigned long long adlen,
                                                  const unsigned char *nsec,
```

const unsigned char *npub,
const unsigned char *k);

تابع crypto_aead_chacha20poly1305_encrypt_detached() پیام m را با کلید k و نانس npub رمز می‌کند. پیام رمزشده که دارای طول برابر با پیام فاش می‌باشد، در c ذخیره می‌شود.

این تابع همچنین یک برچسب که متن رمزشده و اطلاعات اضافی ad به طول adlen را احراز اصالت می‌کند، محاسبه و تولید می‌کند. این برچسب در mac ذخیره می‌شود و دارای اندازه crypto_aead_chacha20poly1305_ABYTES بایت می‌باشد.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

int crypto_aead_chacha20poly1305_decrypt_detached(unsigned char *m,
unsigned char *nsec,
const unsigned char *c,
unsigned long long clen,
const unsigned char *mac,
const unsigned char *ad,
unsigned long long adlen,
const unsigned char *npub,
const unsigned char *k);

تابع crypto_aead_chacha20poly1305_decrypt_detached() بررسی می‌کند که آیا برچسب احراز اصالت mac برای متن رمزشده c به طول clen بایت، کلید k، نانس npub و اطلاعات اضافی ad به طول adlen معتبر است یا خیر.

اگر این برچسب معتبر نبود، تابع عدد 1- را باز می‌گرداند و پردازش اضافی دیگری انجام نمی‌شود.

اگر برچسب معتبر بود، متن رمزشده ابتدا رمزگشایی می‌شود و متن فاش در m ذخیره می‌شود. طول متن رمزگشایی شده با طول متن رمزشده برابر است.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

ثابت‌ها:

- crypto_aead_chacha20poly1305_KEYBYTES
- crypto_aead_chacha20poly1305_NPUBBYTES
- crypto_aead_chacha20poly1305_ABYTES

رمزنگاری: ChaCha20 stream cipher

احراز اصالت: Poly1305 MAC

توجه: برای جلوگیری از استفاده مجدد از نانس، اگر قرار بر این باشد که یک کلید مجدداً مورد استفاده قرار گیرد، توصیه می‌شود به‌جای اینکه یک نانس جدید به صورت تصادفی برای هر پیام تولید شود، همان نانس مرحله قبل یک واحد اضافه شده و مورد استفاده قرار گیرد.

برای جلوگیری از استفاده مجدد از نانس در پروتکل‌های کلاینت-سرور بایستی برای هر کدام از طرف‌ها از کلید متفاوت استفاده شود یا اینکه یک بیت در یک طرف 0 و در طرف دیگر همان بیت 1 شود.

۲-۷-۳-۳ ساختار ChaCha20-Poly1305 – استاندارد IETF

نسخه IETF از ساختار ChaCha20-Poly1305 می‌تواند به طور امن و عملی تعداد نامحدودی از پیام‌ها (۲۹۶ پیام) را رمز کند، البته اندازه هر پیام نباید از یک ترا بایت تجاوز کند.

مثال (حالت ترکیبی):

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define ADDITIONAL_DATA (const unsigned char *) "123456"
#define ADDITIONAL_DATA_LEN 6

unsigned char nonce[crypto_aead_chacha20poly1305_ietf_NPUBBYTES];
unsigned char key[crypto_aead_chacha20poly1305_ietf_KEYBYTES];
unsigned char ciphertext[MESSAGE_LEN + crypto_aead_chacha20poly1305_ietf_ABYTES];
unsigned long long ciphertext_len;

randombytes_buf(key, sizeof key);
randombytes_buf(nonce, sizeof nonce);

crypto_aead_chacha20poly1305_ietf_encrypt(ciphertext, &ciphertext_len,
                          MESSAGE, MESSAGE_LEN,
                          ADDITIONAL_DATA, ADDITIONAL_DATA_LEN,
                          NULL, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
unsigned long long decrypted_len;
if (crypto_aead_chacha20poly1305_ietf_decrypt(decrypted, &decrypted_len,
                          NULL,
                          ciphertext, ciphertext_len,
                          ADDITIONAL_DATA,
                          ADDITIONAL_DATA_LEN,
                          nonce, key) != 0) {
   /* message forged! */
}
```

حالت ترکیبی:

در این حالت، برچسب احراز اصالت و پیام رمزشده با یکدیگر در یک محل ذخیره می‌شوند.

```
int crypto_aead_chacha20poly1305_ietf_encrypt(unsigned char *c,
                          unsigned long long *clen,
                          const unsigned char *m,
                          unsigned long long mlen,
```

```
const unsigned char *ad,
unsigned long long adlen,
const unsigned char *nsec,
const unsigned char *npub,
const unsigned char *k);
```

تابع crypto_aead_chacha20poly1305_ietf_encrypt() پیام m به طول mlen بایت را با استفاده از کلید خصوصی k (از نوع بایت‌های crypto_aead_chacha20poly1305_IETF_KEYBYTES) و نانس npub (از نوع بایت‌های crypto_aead_chacha20poly1305_IETF_NPUBBYTES) رمز می‌کند.

پیام رمزشده و همچنین برچسب احراز اصالت که هم پیام محرمانه m و هم داده‌های غیرمحرمانه ad به طول adlen را احراز اصالت می‌کند، در c ذخیره می‌شوند.
اگر اطلاعات اضافی مورد نیاز نباشد، ad می‌تواند یک اشاره‌گر NULL باشد. در این حالت طول adlen برابر با 0 است.

حداکثر تعداد بایت‌هایی که در c ذخیره می‌شوند و همینطور تعداد بایت‌های واقعی که در clen ذخیره می‌شوند (مگر اینکه lcen یک اشاره‌گر NULL باشد)، برابر با mlen + crypto_aead_chacha20poly1305_ABYTES می‌باشد.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

نانس npub که به صورت عمومی نیز در دسترس است، نباید هیچ موقع در ساخت یک کلید یکسان مجدداً استفاده شود. روش پیشنهادی برای تولید آن استفاده از تابع randombytes_buf() برای اولین پیام و اضافه کردن آن برای هرکدام از پیام‌های زیرمجموعه‌ای است که از یک کلید یکسان استفاده می‌کنند.

```
int crypto_aead_chacha20poly1305_ietf_decrypt(unsigned char *m,
                                              unsigned long long *mlen,
```

```
unsigned char *nsec,
const unsigned char *c,
unsigned long long clen,
const unsigned char *ad,
unsigned long long adlen,
const unsigned char *npub,
const unsigned char *k);
```

تابع () crypto_aead_chacha20poly1305_ietf_decrypt تصدیق می‌کند که آیا متن رمزشده c (که توسط تابع crypto_aead_chacha20poly1305_ietf_encrypt تولید شده است) شامل یک برچسب معتبر می‌باشد که از کلید خصوصی k، نانس npub و اطلاعات اضافی ad به طول adlen بایت استفاده کرده است.

اگر اطلاعات اضافی مورد نیاز نباشد، ad می‌تواند یک اشاره‌گر NULL باشد. در این حالت طول adlen برابر با 0 است.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

اگر نتیجه بررسی موفقیت‌آمیز نباشد، تابع عدد 1- را به عنوان خروجی باز می‌گرداند. در غیر این صورت، عدد 0 بازگردانده می‌شود و پیام رمزگشایی شده در m ذخیره می‌شود. همچنین تعداد بایت‌های واقعی پیام ذخیره شده در m، در mlen ذخیره می‌شود. البته این در حالی است که meln یک اشاره‌گر NULL نباشد.

حداکثر بایت‌هایی که در m می‌تواند ذخیره شود برابر با clen - crypto_aead_chacha20poly1305_IETF_ABYTES است.

حالت مجزا:

بعضی از برنامه‌ها نیاز به ذخیره برچسب احراز اصالت و پیام رمزشده در مکان‌های مختلف دارند. برای این منظور نوع مجزای توابع بالا قابل استفاده هستند.

int crypto_aead_chacha20poly1305_ietf_encrypt_detached(unsigned char *c,

```
                unsigned char *mac,
                unsigned long long *maclen_p,
                const unsigned char *m,
                unsigned long long mlen,
                const unsigned char *ad,
                unsigned long long adlen,
                const unsigned char *nsec,
                const unsigned char *npub,
                const unsigned char *k);
```

تابع crypto_aead_chacha20poly1305_ietf_encrypt_detached() پیام m را با کلید k و نانس npub رمز می‌کند. پیام رمز شده که دارای طول مشابه با پیام اصلی و فاش می‌باشد در c ذخیره می‌شود.

همچنین این تابع یک برچسب که متن رمزشده و اطلاعات اضافی ad به طول adlen را احراز اصالت می‌کند، محاسبه و تولید می‌کند. این برچسب در mac ذخیره می‌شود و دارای اندازه crypto_aead_chacha20poly1305_IETF_ABYTES بایت می‌باشد.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

```
int crypto_aead_chacha20poly1305_ietf_decrypt_detached(unsigned char *m,
                unsigned char *nsec,
                const unsigned char *c,
                unsigned long long clen,
                const unsigned char *mac,
                const unsigned char *ad,
                unsigned long long adlen,
                const unsigned char *npub,
                const unsigned char *k);
```

تابع crypto_aead_chacha20poly1305_ietf_decrypt_detached () بررسی می‌کند که آیا برچسب احراز اصالت mac برای متن رمزشده c با طول clen بایت، کلید k، نانس npub و اطلاعات مازاد ad با طول adlen معتبر است یا خیر.

اگر این برچسب معتبر نبود، تابع عدد 1- را باز می‌گرداند و پردازش اضافی دیگری انجام نمی‌شود.

اگر برچسب معتبر بود، متن رمزشده ابتدا رمزگشایی می‌شود و متن فاش در m ذخیره می‌شود. طول متن رمزگشایی شده با طول متن رمزشده برابر است.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

ثابت‌ها:

- crypto_aead_chacha20poly1305_IETF_ABYTES

و از sodium نسخه 1.0.9 به بعد:

- crypto_aead_chacha20poly1305_IETF_KEYBYTES
- crypto_aead_chacha20poly1305_IETF_NPUBBYTES

در نسخه‌های قدیمی‌تر، باید از crypto_aead_chacha20poly1305_KEYBYTES و crypto_aead_chacha20poly1305_NPUBBYTES استفاده شود. تنها تفاوتی که در میان ثابت‌ها بین نسخه اصلی و نسخه IETF وجود دارد، اندازه نانس می‌باشد.

جزئیات الگوریتم:

- Encryption: ChaCha20 stream cipher
- Authentication: Poly1305 MAC

توجه: برای جلوگیری از استفاده مجدد از نانس، اگر قرار بر این باشد که یک کلید مجدداً مورد استفاده قرار گیرد، توصیه می‌شود به‌جای اینکه یک نانس جدید به صورت تصادفی برای هر پیام تولید شود، همان نانس مرحله قبل یک واحد اضافه شده و مورد استفاده قرار گیرد.

برای جلوگیری از استفاده مجدد از نانس در پروتکل‌های کلاینت-سرور یا برای هر کدام از طرف‌ها از کلید متفاوت استفاده شود و یا اینکه یک بیت در یک طرف 0 شود و در طرف دیگر همان بیت 1 شود.

۲-۷-۳-۴ رمزنگاری احراز اصالت شده همراه اطلاعات اضافی با استفاده از AES-GCM

مثال (حالت ترکیبی) :

```
#include <sodium.h>

#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define ADDITIONAL_DATA (const unsigned char *) "123456"
#define ADDITIONAL_DATA_LEN 6

unsigned char nonce[crypto_aead_aes256gcm_NPUBBYTES];
unsigned char key[crypto_aead_aes256gcm_KEYBYTES];
unsigned char ciphertext[MESSAGE_LEN + crypto_aead_aes256gcm_ABYTES];
unsigned long long ciphertext_len;

sodium_init();
if (crypto_aead_aes256gcm_is_available() == 0) {
   abort(); /* Not available on this CPU */
}

randombytes_buf(key, sizeof key);
randombytes_buf(nonce, sizeof nonce);

crypto_aead_aes256gcm_encrypt(ciphertext, &ciphertext_len,
                              MESSAGE, MESSAGE_LEN,
                              ADDITIONAL_DATA, ADDITIONAL_DATA_LEN,
                              NULL, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
unsigned long long decrypted_len;
```

```
if (ciphertext_len < crypto_aead_aes256gcm_ABYTES ||
    crypto_aead_aes256gcm_decrypt(decrypted, &decrypted_len,
                                  NULL,
                                  ciphertext, ciphertext_len,
                                  ADDITIONAL_DATA,
                                  ADDITIONAL_DATA_LEN,
                                  nonce, key) != 0) {
    /* message forged! */
}
```

این فرآیند شامل مراحل زیر می‌باشد:

- رمزنگاری پیام با یک کلید و نانس برای حفظ محرمانگی آن
- محاسبه برچسب احراز اصالت (این امر برای حصول اطمینان از عدم دستکاری پیام می‌باشد).

یک نمونه از اطلاعات اضافی مورد نظر می‌تواند داده‌های مربوط به یک پروتکل خاص درباره پیام باشد، مانند طول و نحوه کدگذاری آن.

این عملیات همچنین می‌تواند به عنوان MAC به همراه یک پیام خالی استفاده شود.

فرآیند رمزگشایی به هیچ عنوان قبل از بررسی و تصدیق پیام انجام نمی‌شود.

محدودیت‌ها:

پیاده‌سازی فعلی از این حالت، سخت‌افزاری و نیازمند اضافه‌کردن سخت‌افزار Intel SSSE3 و همچنین دستورات aesni و pclmul می‌باشد. در حال حاضر روشی به غیر از روش سخت‌افزاری برای پیاده‌سازی این حالت وجود ندارد.

اگر ویژگی portability مد نظر باشد، باید از ChaCha20-Poly1305 استفاده شود.

قبل از استفاده از تابع زیر، برای بررسی پشتیبانی سخت‌افزار سیستم از AES، از تابع زیر استفاده می‌کنیم:

```
int crypto_aead_aes256gcm_is_available(void);
```

اگر CPU فعلی از پیاده‌سازی AES256-GCM پشتیبانی کند، تابع عدد 1 را باز می‌گرداند و در غیر این صورت عدد 0 بازگردانده می‌شود. ضمن اینکه قبل از فراخوانی این تابع، باید برای مقداردهی اولیه، تابع sodium_init() فراخوانی شود.

حالت ترکیبی:

در این حالت برچسب احراز اصالت و پیام رمزشده در یک محل ذخیره می‌شوند.

```
int crypto_aead_aes256gcm_encrypt(unsigned char *c,
                                  unsigned long long *clen,
                                  const unsigned char *m,
                                  unsigned long long mlen,
                                  const unsigned char *ad,
                                  unsigned long long adlen,
                                  const unsigned char *nsec,
                                  const unsigned char *npub,
                                  const unsigned char *k);
```

تابع crypto_aead_aes256gcm_encrypt () پیام m به طول mlen بایت را با استفاده از کلید خصوصی k (از نوع بایت‌های crypto_aead_aes256gcm_KEYBYTES) و نانس npub (از نوع بایت‌های crypto_aead_aes256gcm_NPUBBYTES) رمز می‌کند.

پیام رمزشده و همچنین برچسب احراز اصالت که هم پیام محرمانه m و هم داده‌های غیرمحرمانه ad به طول adlen را احراز اصالت می‌کند، در c ذخیره می‌شوند.

ad می‌تواند یک اشاره‌گر NULL باشد و اگر اطلاعات اضافی مورد نیاز نباشد، اندازه adlen نیز می‌تواند برابر 0 باشد.

حداکثر تعداد بایت‌هایی که در c ذخیره می‌شوند و همینطور تعداد بایت‌های واقعی که در clen ذخیره می‌شوند (مگر اینکه clen یک اشاره‌گر NULL باشد)، برابر با mlen + crypto_aead_aes256gcm_ABYTES می‌باشد.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

نانس npub که به صورت عمومی نیز در دسترس است، نباید هیچ موقع در ساخت یک کلید یکسان مجدداً استفاده شود. روش پیشنهادی برای تولید آن استفاده از تابع randombytes_buf() برای اولین پیام و اضافه کردن آن برای هرکدام از پیام‌های زیرمجموعه‌ای است که از یک کلید یکسان استفاده می‌کنند.

```
int crypto_aead_aes256gcm_decrypt(unsigned char *m,
                                  unsigned long long *mlen_p,
                                  unsigned char *nsec,
                                  const unsigned char *c,
                                  unsigned long long clen,
                                  const unsigned char *ad,
                                  unsigned long long adlen,
                                  const unsigned char *npub,
                                  const unsigned char *k);
```

تابع crypto_aead_aes256gcm_decrypt () تصدیق می‌کند که آیا متن رمزشده c (که توسط تابع crypto_aead_aes256gcm_encrypt () تولید شده است) شامل یک برچسب معتبر می‌باشد که از کلید خصوصی k، نانس npub و اطلاعات اضافی ad به طول adlen بایت استفاده کرده است.

اگر اطلاعات اضافی مورد نیاز نباشد، ad می‌تواند یک اشاره‌گر NULL باشد.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

اگر نتیجه بررسی موفقیت‌آمیز نباشد، تابع عدد 1- را به عنوان خروجی باز می‌گرداند. در غیر این صورت، عدد 0 بازگردانده می‌شود و پیام رمزگشایی شده در m ذخیره می‌شود. همچنین تعداد بایت‌های واقعی پیام ذخیره شده در m، در mlen ذخیره می‌شود. البته این در حالی است که meln یک اشاره‌گر NULL نباشد.

حداکثر بایت‌هایی که در m می‌تواند ذخیره شود برابر با crypto_aead_aes256gcm_ABYTES - clen است.

حالت مجزا:

بعضی از برنامه‌ها ممکن است که نیاز به ذخیره پیام رمزشده و برچسب احراز اصالت در دو مکان مختلف داشته باشند. در این مورد، نوع مجزای تابع گفته شده در قسمت قبل استفاده می‌شود.

```
int crypto_aead_aes256gcm_encrypt_detached(unsigned char *c,
                                           unsigned char *mac,
                                           unsigned long long *maclen_p,
                                           const unsigned char *m,
                                           unsigned long long mlen,
                                           const unsigned char *ad,
                                           unsigned long long adlen,
                                           const unsigned char *nsec,
                                           const unsigned char *npub,
                                           const unsigned char *k);
```

تابع () crypto_aead_aes256gcm_encrypt_detached پیام m به طول mlen بایت را با استفاده از کلید خصوصی k (از نوع بایت‌های crypto_aead_aes256gcm_KEYBYTES) و نانس npub (از نوع بایت‌های crypto_aead_aes256gcm_NPUBBYTES) رمز می‌کند.

پیام رمزشده و همچنین برچسب احراز اصالت که هم پیام محرمانه m و هم داده‌های غیرمحرمانه ad به طول adlen را احراز اصالت می‌کند، در c ذخیره می‌شوند.

ad می‌تواند یک اشاره‌گر NULL باشد و اگر اطلاعات اضافی مورد نیاز نباشد، اندازه adlen نیز می‌تواند برابر 0 باشد.

بایت‌های crypto_aead_aes256gcm_ABYTES در mac ذخیره می‌شوند. همچنین اگر maclen_p یک اشاره‌گر NULL نباشد، تعداد بایت‌های واقعی مورد نیاز برای احراز اصالت در maclen_p ذخیره می‌شوند.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

ضمناً این تابع همیشه عدد 0 را به عنوان خروجی باز می‌گرداند.

```
int crypto_aead_aes256gcm_decrypt_detached(unsigned char *m,
                                           unsigned char *nsec,
                                           const unsigned char *c,
                                           unsigned long long clen,
                                           const unsigned char *mac,
                                           const unsigned char *ad,
                                           unsigned long long adlen,
                                           const unsigned char *npub,
                                           const unsigned char *k);
```

تابع () crypto_aead_aes256gcm_decrypt_detached بررسی می‌کند که آیا برچسب احراز اصالت mac برای متن رمزشده c با طول clen بایت، کلید k، نانس npub و اطلاعات مازاد ad با طول adlen معتبر است یا خیر.

clen تعداد بایت‌های متن رمزشده بر حسب بایت می‌باشد.

اگر اطلاعات اضافی مورد نیاز نباشد، ad می‌تواند یک اشاره‌گر NULL باشد.

nsec در این ساختار استفاده نمی‌شود و باید همیشه NULL باشد.

اگر این برچسب معتبر نباشد، تابع عدد 1- را باز می‌گرداند.

اگر نتیجه بررسی موفقیت‌آمیز نباشد، تابع عدد 1- را به عنوان خروجی باز می‌گرداند. در غیر این صورت، عدد 0 بازگردانده می‌شود و پیام رمزگشایی شده در m ذخیره می‌شود. همچنین طول متن رمزگشایی شده با طول متن رمزشده برابر است.

ثابت‌ها:

- crypto_aead_aes256gcm_KEYBYTES
- crypto_aead_aes256gcm_NPUBBYTES
- crypto_aead_aes256gcm_ABYTES

توجه: طول نانس 96 بیت است. برای جلوگیری از استفاده مجدد از نانس، اگر قرار بر این باشد که یک کلید مجدداً مورد استفاده قرار گیرد، توصیه می‌شود به‌جای اینکه یک نانس جدید به صورت تصادفی برای هر پیام تولید شود، همان نانس مرحله قبل یک واحد اضافه شده و مورد استفاده قرار گیرد.

برای جلوگیری از استفاده مجدد از نانس در پروتکل‌های کلاینت-سرور، بایستی برای هر کدام از طرف‌ها از کلید متفاوت استفاده شود و یا اینکه یک بیت در یک طرف 0 شود و در طرف دیگر همان بیت 1 شود.

توصیه می‌شود که پیام‌های بزرگتر از 2 Gb به تکه‌های کوچکتر تقسیم شوند.

۲-۷-۳-۵ AES256-GCM همراه پیش‌محاسبه

برنامه‌هایی که چندین پیام را با استفاده از یک کلید یکسان رمز می‌کنند، می‌توانند از طریق واسط پیش‌محاسبه و انتشار کلید AES فقط برای یک بار، باعث کمی افزایش در سرعت برنامه شوند.

```
int crypto_aead_aes256gcm_beforenm(crypto_aead_aes256gcm_state *ctx_,
                                   const unsigned char *k);
```

تابع crypto_aead_aes256gcm_beforenm() با انتشار کلید k، ctx را initialize می‌کند. این تابع همیشه عدد 0 را به عنوان خروجی باز می‌گرداند.

16 بایت به منظور هم‌ترازی، برای آدرس ctx مورد نیاز است. اندازه این مقدار می‌تواند با استفاده از sizeof (crypto_aead_aes256gcm_state) یا crypto_aead_aes256gcm_statebytes() به‌دست آید.

حالت ترکیبی به همراه پیش‌محاسبه:

```
int crypto_aead_aes256gcm_encrypt_afternm(unsigned char *c,
                                          unsigned long long *clen_p,
                                          const unsigned char *m,
                                          unsigned long long mlen,
                                          const unsigned char *ad,
                                          unsigned long long adlen,
                                          const unsigned char *nsec,
                                          const unsigned char *npub,
                                          const crypto_aead_aes256gcm_state *ctx_);

int crypto_aead_aes256gcm_decrypt_afternm(unsigned char *m,
                                          unsigned long long *mlen_p,
                                          unsigned char *nsec,
                                          const unsigned char *c,
                                          unsigned long long clen,
                                          const unsigned char *ad,
                                          unsigned long long adlen,
                                          const unsigned char *npub,
                                          const crypto_aead_aes256gcm_state *ctx_);
```

توابع crypto_aead_aes256gcm_encrypt_afternm() و crypto_aead_aes256gcm_decrypt_afternm() همانند crypto_aead_aes256gcm_encrypt() و crypto_aead_aes256gcm_decrypt() می‌باشند، البته با این تفاوت که به‌جای دریافت کلید، یک متن مقداردهی شده (ctx) دریافت می‌کند.

حالت مجزا به همراه پیش‌محاسبه:

```
int crypto_aead_aes256gcm_encrypt_detached_afternm(unsigned char *c,
                                                   unsigned char *mac,
                                                   unsigned long long *maclen_p,
                                                   const unsigned char *m,
                                                   unsigned long long mlen,
                                                   const unsigned char *ad,
                                                   unsigned long long adlen,
                                                   const unsigned char *nsec,
                                                   const unsigned char *npub,
                                                   const crypto_aead_aes256gcm_state *ctx_);

int crypto_aead_aes256gcm_decrypt_detached_afternm(unsigned char *m,
                                                   unsigned char *nsec,
                                                   const unsigned char *c,
                                                   unsigned long long clen,
                                                   const unsigned char *mac,
                                                   const unsigned char *ad,
                                                   unsigned long long adlen,
                                                   const unsigned char *npub,
                                                   const crypto_aead_aes256gcm_state *ctx_)
```

توابع crypto_aead_aes256gcm_encrypt_detached_afternm() و crypto_aead_aes256gcm_decrypt_detached_afternm() همانند crypto_aead_aes256gcm_encrypt_detached() و crypto_aead_aes256gcm_decrypt_detached() می‌باشند، البته با این تفاوت که به‌جای دریافت کلید، یک متن مقداردهی شده (ctx) دریافت می‌کنند.

ثابت‌ها:

- crypto_aead_aes256gcm_KEYBYTES
- crypto_aead_aes256gcm_NPUBBYTES
- crypto_aead_aes256gcm_ABYTES

نوع داده:

- crypto_aead_aes256gcm_state

توجه: طول نانس 96 بیت است. برای جلوگیری از استفاده مجدد از نانس، اگر قرار بر این باشد که یک کلید مجدداً مورد استفاده قرار گیرد، توصیه می‌شود به‌جای اینکه یک نانس جدید به صورت تصادفی برای هر پیام تولید شود، همان نانس مرحله قبل یک واحد اضافه شده و مورد استفاده قرار گیرد.

برای جلوگیری از استفاده مجدد از نانس در پروتکل‌های کلاینت-سرور، بایستی برای هر کدام از طرف‌ها از کلید متفاوت استفاده شود و یا اینکه یک بیت در یک طرف 0 شود و در طرف دیگر همان بیت 1 شود.

توصیه می‌شود که پیام‌های بزرگتر از 2 Gb به تکه‌های کوچکتر تقسیم شوند.

۲-۸ رمزنگاری کلید عمومی

۲-۸-۱ رمزنگاری احراز اصالت شده

مثال:

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define CIPHERTEXT_LEN (crypto_box_MACBYTES + MESSAGE_LEN)

unsigned char alice_publickey[crypto_box_PUBLICKEYBYTES];
unsigned char alice_secretkey[crypto_box_SECRETKEYBYTES];
crypto_box_keypair(alice_publickey, alice_secretkey);

unsigned char bob_publickey[crypto_box_PUBLICKEYBYTES];
unsigned char bob_secretkey[crypto_box_SECRETKEYBYTES];
crypto_box_keypair(bob_publickey, bob_secretkey);

unsigned char nonce[crypto_box_NONCEBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];
randombytes_buf(nonce, sizeof nonce);
```

```
if (crypto_box_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce,
                    bob_publickey, alice_secretkey) != 0) {
    /* error */
}

unsigned char decrypted[MESSAGE_LEN];
if (crypto_box_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce,
                         alice_publickey, bob_secretkey) != 0) {
    /* message for Bob pretending to be from Alice has been forged! */
}
```

با استفاده از رمزنگاری احراز اصالت شده کلید عمومی، طرف A می‌تواند یک پیام محرمانه را فقط برای طرف B رمز کند، به گونه‌ای که غیر از B کسی توانایی رمزگشایی آن را نداشته باشد. A پیام را با کلید عمومی B رمز و ارسال می‌کند.

طرف B با استفاده از کلید عمومی طرف A و قبل از رمزگشایی پیام، می‌تواند تصدیق کند که پیام دریافت شده واقعاً از سوی A ارسال شده است.

طرف B تنها به کلید عمومی طرف A، نانس استفاده شده و متن رمزشده نیاز دارد. ضمن اینکه هیچکدام از طرفین نباید کلید خصوصی خود را با دیگران به اشتراک بگذارند.

در این عملیات، نانس فاش است و نیازی به محرمانه بودن آن وجود ندارد، ولی هر نانس منحصر به فرد برای هر زوج کلید عمومی و خصوصی باید تنها در یک فراخوانی از تابع crypto_box_open_easy() استفاده شود.

یک روش آسان برای تولید نانس، استفاده از تابع randombytes_buf() می‌باشد. با توجه به اندازه نانس‌ها، احتمال حدس نانس صحیح و یا به عبارت دیگر احتمال تصادم تصادفی نانس‌ها بسیار ناچیز است. برای بعضی از برنامه‌ها اگر هدف، استفاده از نانس به منظور شناسایی پیام‌های گم شده یا دور انداختن پیام‌های تکراری باشد، استفاده از یک شمارنده ساده که به طور مرتب افزایش پیدا می‌کند به عنوان نانس کفایت می‌کند.

هنگام استفاده از این روش، باید از عدم استفاده مجدد از یک نانس اطمینان حاصل نمود. برای مثال، حالتی که چندین thread و یا حتی host، اقدام به تولید پیام با استفاده از یک کلید یکسان می‌کنند.
این فرآیند یک احراز اصالت دوطرفه را فراهم می‌کند. سناریوی معمول، امن‌سازی ارتباط بین یک سرور (که کلید عمومی آن مشخص می‌باشد) و کلاینت‌هایی می‌باشد که به طور ناشناس به آن متصل هستند.

تولید جفت کلید:

```
int crypto_box_keypair(unsigned char *pk, unsigned char *sk);
```

تابع crypto_box_keypair() به طور تصادفی یک کلید خصوصی و کلیدعمومی متناظر با آن را تولید می‌کند. کلیدعمومی در pk (از نوع بایت‌های crypto_box_PUBLICKEYBYTES) و کلیدخصوصی در sk (ازنوع بایت‌های crypto_box_SECRETKEYBYTES) ذخیره می‌شود.

```
int crypto_box_seed_keypair(unsigned char *pk, unsigned char *sk,
                            const unsigned char *seed);
```

با استفاده از تابع crypto_box_seed_keypair()، می‌توان یک جفت کلید به صورت قطعی از یک کلید seed (از نوع بایت‌های crypto_box_SEEDBYTES) مشتق گرفت.

```
int crypto_scalarmult_base(unsigned char *q, const unsigned char *n);
```

علاوه بر این، تابع crypto_scalarmult_base() می‌تواند برای محاسبه کلید عمومی استفاده شود (با توجه به آن که کلید خصوصی قبلاً توسط تابع crypto_box_keypair() تولید شده باشد).

```
unsigned char pk[crypto_box_PUBLICKEYBYTES];
crypto_scalarmult_base(pk, sk);
```

حالت ترکیبی:

در این حالت، برچسب احراز اصالت و پیام رمزشده با هم در یک محل ذخیره می‌شوند.

```
int crypto_box_easy(unsigned char *c, const unsigned char *m,
                    unsigned long long mlen, const unsigned char *n,
                    const unsigned char *pk, const unsigned char *sk);
```

تابع crypto_box_easy() پیام m به طول mlen بایت را با استفاده از کلیدعمومی طرف گیرنده، pk، کلیدخصوصی فرستنده، sk، و نانس n رمز می‌کند.

n باید از نوع بایت‌های crypto_box_NONCEBYTES باشد.

c باید حداقل دارای طول crypto_box_MACBYTES + mlen باشد.

این تابع برچسب احراز اصالت دارای طول crypto_box_MACBYTES بایت می‌باشد و بلافاصله پس از آن پیام رمزشده که دارای طول mlen و مشابه با طول پیام فاش می‌باشد را در c می‌نویسد.

```
int crypto_box_open_easy(unsigned char *m, const unsigned char *c,
                         unsigned long long clen, const unsigned char *n,
                         const unsigned char *pk, const unsigned char *sk);
```

تابع crypto_box_open_easy() متن رمزشده توسط تابع crypto_box_open_easy() را تصدیق و رمزگشایی می‌کند.

c یک اشاره‌گر به ترکیب برچسب احراز اصالت و پیام رمزشده می‌باشد که توسط تابع crypto_box_easy() تولید شده است. clen طول این ترکیب برچسب احراز اصالت و پیام رمزشده می‌باشد. به عبارت دیگر clen تعداد بایت‌های نوشته شده توسط تابع crypto_box_easy() می‌باشد که برابر با crypto_box_MACBYTES به علاوه طول پیام می‌باشد.

باید نانس n با نانس استفاده شده برای رمزنگاری و احراز اصالت پیام یکسان باشد. pk کلیدعمومی فرستنده پیام می‌باشد که آن را رمز می‌کند. sk کلیدخصوصی گیرنده پیام می‌باشد.

اگر نتیجه احراز اصالت موفقیت‌آمیز نباشد، تابع عدد 1- را باز می‌گرداند. در غیر این صورت عدد 0 بازگردانده می‌شود و پیام رمزگشایی شده در m ذخیره می‌شود.

حالت مجزا:

در این حالت برچسب احراز اصالت و پیام رمزشده در مکان‌های مختلف ذخیره می‌شوند.

```
int crypto_box_detached(unsigned char *c, unsigned char *mac,
                        const unsigned char *m,
                        unsigned long long mlen,
                        const unsigned char *n,
                        const unsigned char *pk,
                        const unsigned char *sk);
```

این تابع پیام m با طول mlen بایت را با استفاده از نانس n و کلیدخصوصی sk، برای گیرنده‌ای که دارای کلیدعمومی pk است، رمز می‌کند و پیام رمزشده را در c ذخیره می‌کند. همچنین دقیقاً mlen بایت در c ذخیره می‌شود.

برچسب احراز اصالت که دارای طول crypto_box_MACBYTES می‌باشد در mac ذخیره می‌شود.

```
int crypto_box_open_detached(unsigned char *m,
                             const unsigned char *c,
                             const unsigned char *mac,
                             unsigned long long clen,
                             const unsigned char *n,
                             const unsigned char *pk,
                             const unsigned char *sk);
```

تابع crypto_box_open_detached () پیام رمزشده c با طول clen را با استفاده از کلیدخصوصی sk مربوط به گیرنده و کلیدعمومی فرستنده، pk، تصدیق و رمزگشایی می‌کند.

clen شامل برچسب احراز اصالت نمی‌باشد، بنابراین طول آن همانند متن فاش می‌باشد.

پس از بررسی و تصدیق mac و معتبر شناخته شدن آن به عنوان یک برچسب احراز اصالت معتبر با فرض داشتن نانس n و کلید k، متن فاش در m ذخیره می‌شود.

اگر نتیجه احراز اصالت موفقیت‌آمیز باشد عدد 0 و در غیر این صورت عدد 1- بازگردانده می‌شود.

واسط پیش‌محاسبه:

برنامه‌هایی که چندین پیام را به یک گیرنده خاص ارسال می‌کنند و یا اینکه چندین پیام را از یک فرستنده دریافت می‌کنند، می‌توانند با محاسبه کلید به اشتراک گذاشته شده برای تنها یک بار و استفاده از آن در عملیات‌های بعدی باعث افزایش سرعت شوند.

```
int crypto_box_beforenm(unsigned char *k, const unsigned char *pk,
                        const unsigned char *sk);
```

تابع crypto_box_beforenm() کلیدخصوصی مشترک را با توجه به دراختیار داشتن کلیدعمومی pk و کلیدخصوصی sk محاسبه می‌کند و نتیجه را در k ذخیره می‌کند (از نوع بایت‌های crypto_box_BEFORENMBYTES).

```
int crypto_box_easy_afternm(unsigned char *c, const unsigned char *m,
                            unsigned long long mlen, const unsigned char *n,
                            const unsigned char *k);

int crypto_box_open_easy_afternm(unsigned char *m, const unsigned char *c,
                                 unsigned long long clen, const unsigned char *n,
                                 const unsigned char *k);
```

```
int crypto_box_detached_afternm(unsigned char *c, unsigned char *mac,
                                const unsigned char *m, unsigned long long mlen,
                                const unsigned char *n, const unsigned char *k);

int crypto_box_open_detached_afternm(unsigned char *m, const unsigned char *c,
                                     const unsigned char *mac,
                                     unsigned long long clen, const unsigned char *n,
                                     const unsigned char *k);
```

نوع afternm _ از تابع توضیح داده شده، به جای دریافت یک زوج کلید، یک کلیدخصوصی مشترک (k) را به عنوان ورودی دریافت می‌کند.

مانند هر کلید رمزنگاری دیگر، این کلید از پیش محاسبه شده باید بعد از استفاده و نیاز نداشتن به آن، به سرعت از روی حافظه پاک شود (برای مثال از طریق sodium_memzero()).

ثابت‌ها:

- crypto_box_PUBLICKEYBYTES
- crypto_box_SECRETKEYBYTES
- crypto_box_MACBYTES
- crypto_box_NONCEBYTES
- crypto_box_SEEDBYTES
- crypto_box_BEFORENMBYTES

جزئیات الگوریتم:

- Key exchange: X25519
- Encryption: XSalsa20 stream cipher
- Authentication: Poly1305 MAC

توجه: تابع crypto_box () یک اشاره‌گر به فضای ۳۲ بایتی قبل از پیام دریافت می‌کند و متن رمزشده را در مکانی معادل ۱۶ بایت پس از اشاره‌گر مقصد ذخیره می‌کند. همچنین ۱۶ بایت ابتدایی نیز با 0 مجدداً بازنویسی می‌شود.

تابع () crypto_box_open یک اشاره‌گر به فضای ۱۶ بایتی قبل از متن رمزشده دریافت می‌کند و پیام را در مکانی معادل ۳۲ بایت پس از اشاره‌گر مقصد ذخیره می‌کند. همچنین ۳۲ بایت ابتدایی با 0 مجدداً بازنویسی می‌شود.

۲-۸-۲ امضاهای کلید عمومی

مثال: (حالت ترکیبی)

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char pk[crypto_sign_PUBLICKEYBYTES];
unsigned char sk[crypto_sign_SECRETKEYBYTES];
crypto_sign_keypair(pk, sk);

unsigned char signed_message[crypto_sign_BYTES + MESSAGE_LEN];
unsigned long long signed_message_len;

crypto_sign(signed_message, &signed_message_len,
        MESSAGE, MESSAGE_LEN, sk);

unsigned char unsigned_message[MESSAGE_LEN];
unsigned long long unsigned_message_len;
if (crypto_sign_open(unsigned_message, &unsigned_message_len,
                signed_message, signed_message_len, pk) != 0) {
   /* Incorrect signature! */
}
```

مثال: (حالت مجزا)

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char pk[crypto_sign_PUBLICKEYBYTES];
```

```
unsigned char sk[crypto_sign_SECRETKEYBYTES];
crypto_sign_keypair(pk, sk);

unsigned char sig[crypto_sign_BYTES];

crypto_sign_detached(sig, NULL, MESSAGE, MESSAGE_LEN, sk);

if (crypto_sign_verify_detached(sig, MESSAGE, MESSAGE_LEN, pk) != 0) {
   /* Incorrect signature! */
}
```

در این فرآیند، امضاء کننده یک جفت کلید تولید می‌کند:

- یک کلیدخصوصی، که برای اضافه کردن امضاء به هر تعداد پیام استفاده می‌شود.
- یک کلیدعمومی، که هر کسی می‌تواند برای بررسی و تصدیق اینکه امضای الحاق شده به پیام واقعاً از سوی سازنده کلیدعمومی بوده است، استفاده کند.

تصدیق‌کننده‌ها قبل از بررسی و تصدیق پیام‌های امضاء شده باید کلید عمومی را بدانند و به طور قطعی به آن اعتماد داشته باشند.

توجه: این فرآیند با رمزنگاری احراز اصالت شده متفاوت است. الحاق کردن امضا، نحوه نمایش پیام را تغییر نمی‌دهد.

تولید جفت کلید:

```
int crypto_sign_keypair(unsigned char *pk, unsigned char *sk);
```

تابع crypto_sign_keypair() به طور تصادفی یک کلید خصوصی و کلید عمومی متناظر با آن را تولید می‌کند. کلید عمومی در pk (از نوع بایت‌های crypto_sign_PUBLICKEYBYTES) و کلید خصوصی در sk (از نوع بایت‌های crypto_sign_SECRETKEYBYTES) ذخیره می‌شود.

```
int crypto_sign_seed_keypair(unsigned char *pk, unsigned char *sk,
```

const unsigned char *seed);

با استفاده از تابع crypto_sign_seed_keypair() می‌توان به طور قطعی یک جفت کلید از یک کلید seed (از نوع بایت‌های crypto_sign_SEEDBYTES) مشتق گرفت.

حالت ترکیبی:

int crypto_sign(unsigned char *sm, unsigned long long *smlen,
const unsigned char *m, unsigned long long mlen,
const unsigned char *sk);

تابع crypto_sign() با استفاده از کلیدخصوصی sk یک امضاء به پیام m به طول mlen بایت اضافه می‌کند. پیام امضاء شده که شامل امضاء به علاوه یک کپی ساده از پیام می‌باشد، در sm ذخیره می‌شود و دارای طول crypto_sign_BYTES + mlen بایت می‌باشد.

اگر smlen یک اشاره‌گر NULL نباشد، طول واقعی پیام امضاء شده در smlen ذخیره می‌شود.

int crypto_sign_open(unsigned char *m, unsigned long long *mlen,
const unsigned char *sm, unsigned long long smlen,
const unsigned char *pk);

تابع crypto_sign_open() بررسی می‌کند که آیا پیام m به طول smlen بایت، یک امضای معتبر برای کلیدعمومی pk دارد یا خیر.

اگر امضاء معتبر نباشد، تابع عدد 1- را باز می‌گرداند. در صورت معتبر بودن امضاء، تابع عدد 0 را باز می‌گرداند و پیام بدون امضاء را در m و طول آن را در meln ذخیره می‌کند (البته اگر mlen یک اشاره‌گر NULL نباشد).

حالت مجزا:

در این حالت، امضاء بدون ضمیمه یک کپی از پیام اصلی به آن ذخیره می‌شود.

```
int crypto_sign_detached(unsigned char *sig, unsigned long long *siglen,
                         const unsigned char *m, unsigned long long mlen,
                         const unsigned char *sk);
```

تابع crypto_sign_detached() پیام m به طول mlen بایت را با استفاده از کلیدخصوصی sk امضاء می‌کند و امضاء را در sig ذخیره می‌کند. sig می‌تواند حداکثر دارای طول crypto_sign_BYTES بایت باشد.

اگر siglen، NULL نباشد، طول واقعی امضاء در آن ذخیره می‌شود. البته ایرادی ندارد که siglen را در نظر نگیریم و همیشه فرض کنیم که امضاء دارای طول crypto_sign_BYTES بایت می‌باشد. همچنین امضاهای با طول کم در صورت نیاز با اضافه کردن 0 به آن تصحیح می‌شوند.

```
int crypto_sign_verify_detached(const unsigned char *sig,
                                const unsigned char *m,
                                unsigned long long mlen,
                                const unsigned char *pk);
```

تابع crypto_sign_verify_detached() با استفاده pk (کلیدعمومی امضاء کننده)، بررسی می‌کند که آیا sig یک امضای معتبر برای پیام m به طول meln می‌باشد یا خیر. اگر امضاء معتبر نباشد، تابع عدد 1- و اگر معتبر باشد، عدد 0 را باز می‌گرداند.

استخراج seed و کلیدعمومی از کلیدخصوصی:

کلیدخصوصی در واقع شامل seed (یا یک seed تصادفی و یا خروجی تابع crypto_sign_seed_keypair()) و کلیدعمومی می‌باشد.

در حالی که کلیدعمومی همیشه می‌تواند از seed مشتق گرفته شود، ولی پیش‌محاسبه‌سازی به صورت قابل توجهی از میزان سیکل‌های CPU هنگام امضاء کردن می‌کاهد.

Sodium دو تابع برای استخراج seed و کلیدعمومی از کلیدخصوصی فراهم می‌کند:

```
int crypto_sign_ed25519_sk_to_seed(unsigned char *seed,
                                   const unsigned char *sk);

int crypto_sign_ed25519_sk_to_pk(unsigned char *pk, const unsigned char *sk);
```

تابع ()crypto_sign_ed25519_sk_to_seed، seed را از کلیدخصوصی sk استخراج و آن را در متغیر seed (از نوع بایت‌های crypto_sign_SEEDBYTES) ذخیره می‌کند.

تابع ()crypto_sign_ed25519_sk_to_pk کلیدعمومی را از کلیدخصوصی sk استخراج و آن را در pk کپی می‌کند (از نوع بایت‌های crypto_sign_PUBLICKEYBYTES).

ثابت‌ها:

- crypto_sign_PUBLICKEYBYTES
- crypto_sign_SECRETKEYBYTES
- crypto_sign_BYTES
- crypto_sign_SEEDBYTES

جزئیات الگوریتم:

- Signature: Ed25519

توجه: توابع ()crypto_sign_verify و ()crypto_sign_verify_detached فقط به منظور بررسی و تصدیق امضاهای محاسبه شده با استفاده از توابع ()crypto_sign و ()crypto_sign_detached طراحی شده‌اند. تابع ()crypto_sign_open اصلی مربوط به NaCl به گونه‌ای طراحی شده است که 64 بایت پس از پیام را بازنویسی می‌کند. ولی در پیاده‌سازی مربوط به libsodium این کار انجام نمی‌شود.

Sealed boxes ۳-۸-۲

مثال:

```
#define MESSAGE (const unsigned char *) "Message"
#define MESSAGE_LEN 7
#define CIPHERTEXT_LEN (crypto_box_SEALBYTES + MESSAGE_LEN)

/* Recipient creates a long-term key pair */
unsigned char recipient_pk[crypto_box_PUBLICKEYBYTES];
unsigned char recipient_sk[crypto_box_SECRETKEYBYTES];
crypto_box_keypair(recipient_pk, recipient_sk);

/* Anonymous sender encrypts a message using an ephemeral key pair
 * and the recipient's public key */
unsigned char ciphertext[CIPHERTEXT_LEN];
crypto_box_seal(ciphertext, MESSAGE, MESSAGE_LEN, recipient_pk);

/* Recipient decrypts the ciphertext */
unsigned char decrypted[MESSAGE_LEN];
if (crypto_box_seal_open(decrypted, ciphertext, CIPHERTEXT_LEN,
                         recipient_pk, recipient_sk) != 0) {
  /* message corrupted or not intended for this recipient */
}
```

هدف: Sealed box به منظور ارسال پیام‌هایی به طور ناشناس به گیرنده بر اساس کلید عمومی آن طراحی شده است.

فقط گیرنده می‌تواند پیام را با استفاده از کلید خصوصی‌اش رمزگشایی کند (البته گیرنده در حالی که می‌تواند صحت پیام رسیده را تصدیق کند، ولی قادر به تصدیق و تشخیص هویت فرستنده نمی‌باشد).

پیام با استفاده از یک جفت کلید یک‌بار مصرف رمز می‌شود، به طوری که بخش خصوصی آن بلافاصله پس از فرآیند رمزنگاری از بین می‌رود.

بدون دانستن کلیدخصوصی استفاده شده برای یک پیام، فرستنده نمی‌تواند آن پیام را رمزگشایی کند و بدون داشتن اطلاعات اضافی، یک پیام نمی‌تواند با مشخصات و هویت فرستنده همبسته شود.

نحوه استفاده:

```
int crypto_box_seal(unsigned char *c, const unsigned char *m,
                    unsigned long long mlen, const unsigned char *pk);
```

تابع crypto_box_seal() ، پیام m به طول mlen را برای گیرنده‌ای که کلیدعمومی آن pk است، رمز می‌کند. متن رمزشده با طول crypto_box_SEALBYTES + mlen بایت در c قرار داده می‌شود.

تابع یک زوج کلید جدید برای هر پیام تولید می‌کند و کلید عمومی را به متن رمزشده الحاق می‌کند. کلید خصوصی مجدداً بازنویسی می‌شود و پس از اتمام وظیفه، این تابع دیگر در دسترس نخواهد بود.

```
int crypto_box_seal_open(unsigned char *m, const unsigned char *c,
                         unsigned long long clen,
                         const unsigned char *pk, const unsigned char *sk);
```

تابع crypto_box_seal_open() متن رمزشده c با طول clen را با استفاده از زوج کلید (pk, sk) رمزگشایی می‌کند و پیام رمزگشایی شده را در m (با طول clen - crypto_box_SEALBYTES) ذخیره می‌کند. زوج کلید با دیگر عملیات‌های crypto_box_* سازگار می‌باشد و با استفاده از توابع crypto_box_keypair() و crypto_box_seed_keypair() می‌توان آن‌ها را ساخت.

این تابع به دلیل آنکه متن رمزشده شامل کلیدعمومی فرستنده می‌باشد، نیازی به آن ندارد.

ثابت‌ها:

- crypto_box_SEALBYTES

جزئیات الگوریتم:

فرمت sealed box به صورت زیر است:

ephemeral_pk || box(m, recipient_pk, ephemeral_sk, nonce=blake2b(ephemeral_pk || recipient_pk))

۹-۲ Hashing

۱-۹-۲ Generic hashing

مثال تک بخشی بدون کلید:

```
#define MESSAGE ((const unsigned char *) "Arbitrary data to hash")
#define MESSAGE_LEN 22

unsigned char hash[crypto_generichash_BYTES];

crypto_generichash(hash, sizeof hash,
                   MESSAGE, MESSAGE_LEN,
                   NULL, 0);
```

مثال تک بخشی همراه کلید:

```
#define MESSAGE ((const unsigned char *) "Arbitrary data to hash")
#define MESSAGE_LEN 22

unsigned char hash[crypto_generichash_BYTES];
unsigned char key[crypto_generichash_KEYBYTES];

randombytes_buf(key, sizeof key);

crypto_generichash(hash, sizeof hash,
                   MESSAGE, MESSAGE_LEN,
                   key, sizeof key);
```

مثال چند بخشی همراه کلید:

```
#define MESSAGE_PART1 \
   ((const unsigned char *) "Arbitrary data to hash")
#define MESSAGE_PART1_LEN 22
```

```
#define MESSAGE_PART2 \
   ((const unsigned char *) "is longer than expected")
#define MESSAGE_PART2_LEN 23

unsigned char hash[crypto_generichash_BYTES];
unsigned char key[crypto_generichash_KEYBYTES];
crypto_generichash_state state;

randombytes_buf(key, sizeof key);

crypto_generichash_init(&state, key, sizeof key, sizeof hash);

crypto_generichash_update(&state, MESSAGE_PART1, MESSAGE_PART1_LEN);
crypto_generichash_update(&state, MESSAGE_PART2, MESSAGE_PART2_LEN);

crypto_generichash_final(&state, hash, sizeof hash);
```

هدف:

این تابع یک اثر انگشت با طول ثابت برای یک پیام طولانی و دلخواه محاسبه می‌کند.

نمونه موارد مورد استفاده:

- بررسی صحت فایل
- ساخت مشخصه‌های یکتا برای شاخص‌سازی اطلاعات طولانی دلخواه

نحوه استفاده:

```
int crypto_generichash(unsigned char *out, size_t outlen,
                       const unsigned char *in, unsigned long long inlen,
                       const unsigned char *key, size_t keylen);
```

تابع crypto_generichash() یک اثرانگشت از پیام m به طول inlen بایت را در out ذخیره می‌کند. اندازه خروجی می‌تواند توسط برنامه انتخاب شود.

کوچکترین اندازه توصیه شده خروجی برابر با crypto_generichash_BYTES می‌باشد. این اندازه موجب می‌شود که به طور عملی تولید دو اثر انگشت مشابه از دو پیام غیر ممکن باشد.

اما برای یک مورد استفاده خاص، این اندازه می‌تواند هر مقداری بین crypto_generichash_BYTES_MIN و crypto_generichash_BYTES_MAX داشته باشد.

Key می‌تواند NULL و بنابراین keylen برابر 0 باشد. در این مورد، پیام همیشه دارای یک اثر انگشت یکسان و مشابه می‌باشد که شبیه توابع ND5 یا SHA-1 است.

البته یک کلید می‌تواند همچنان اختصاصی باشد. یک پیام با توجه به داشتن یک کلید یکسان، همیشه دارای یک اثر انگشت مشابه خواهد بود، ولی نتایج hash مربوط به یک پیام با کلیدهای مختلف متفاوت می‌باشد.

به طور خاص، می‌توان با استفاده از یک کلید در برنامه‌های مختلف به این نتیجه رسید که برنامه‌های مختلف اثر انگشت‌های متفاوتی تولید می‌کنند حتی اگر اطلاعات یکسانی را پردازش کنند.

اندازه توصیه شده برای کلید، crypto_generichash_KEYBYTES بایت می‌باشد. به هر حال، اندازه کلید می‌تواند هر مقداری بین crypto_generichash_KEYBYTES_MIN و crypto_generichash_KEYBYTES_MAX داشته باشد.

```
int crypto_generichash_init(crypto_generichash_state *state,
                            const unsigned char *key,
                            const size_t keylen, const size_t outlen);

int crypto_generichash_update(crypto_generichash_state *state,
                              const unsigned char *in,
                              unsigned long long inlen);
```

```
int crypto_generichash_final(crypto_generichash_state *state,
                             unsigned char *out, const size_t outlen);
```

لازم نیست که پیام به صورت یک تکه‌ای باشد. همچنین عملیات generichash از واسطهای streaming پشتیبانی می‌کند.

تابع crypto_generichash_init() حالت state را با کلید key (که می‌تواند خالی باشد) به طول keylen بایت، به منظور تولید تعداد outlen بایت از خروجی، مقداردهی اولیه می‌کند.

هر تکه از پیام کامل می‌تواند به ترتیب با فراخوانی تابع crypto_generichash_update() پردازش شود.

تابع crypto_generichash_final() عملیات را کامل می‌کند و اثر انگشت نهایی را در out به طول outlen ذخیره می‌کند.

پس از اتمام عملیات تابع crypto_generichash_final()، نباید حالت فعلی در ادامه استفاده شود، مگر اینکه با استفاده از تابع crypto_generichash_init() مجدداً مقداردهی اولیه شود.

این API به طور خاص برای پردازش فایل‌های بسیار بزرگ و اطلاعات stream مفید می‌باشد.

اندازه ساختار state :

طول ساختار crypto_generichash_state، 357 بایت و یا 361 بایت می‌باشد. 64 بایت به منظور هم‌ترازی و برای افزایش کارایی توصیه می‌شود، ولی نیازی به آن نیست. برای حالت‌های تخصیص داینامیک، تابع crypto_generichash_statebytes() اندازه گرد شده مربوط به ساختار را باز می‌گرداند.

```
state = sodium_malloc(crypto_generichash_statebytes());
```

ثابت‌ها:

- crypto_generichash_BYTES
- crypto_generichash_BYTES_MIN
- crypto_generichash_BYTES_MAX
- crypto_generichash_KEYBYTES
- crypto_generichash_KEYBYTES_MIN
- crypto_generichash_KEYBYTES_MAX

نوع داده‌ها:

- crypto_generichash_state

جزئیات الگوریتم: BLAKE2b

۲-۹-۲ عملیات hash مربوط به ورودی‌های کوچک

مثال:

```
#define SHORT_DATA ((const unsigned char *) "Sparkling water")
#define SHORT_DATA_LEN 15

unsigned char hash[crypto_shorthash_BYTES];
unsigned char key[crypto_shorthash_KEYBYTES];

randombytes_buf(key, sizeof key);
crypto_shorthash(hash, SHORT_DATA, SHORT_DATA_LEN, key);
```

هدف:

بسیاری از پیاده‌سازی‌های مربوط به برنامه‌ها و زبان‌های برنامه‌نویسی هنگامی که از توابع hash با امنیت پایین برای تولید جداول hash استفاده می‌کنند، نسبت به حملات منع سرویس آسیب‌پذیر هستند. Sodium برای رفع این مشکل، از تابع crypto_shorthash() استفاده می‌کند. این تابع دارای خروجی کوچک اما غیرقابل پیش‌بینی (بدون دانستن کلیدخصوصی) می‌باشد. این تابع برای ورودی‌های کوچک بهینه‌سازی شده است. خروجی این تابع فقط 64 بیت است. بنابراین در برابر برخورد (collision) مقاوم نیست.

موارد استفاده:

- جداول hash
- ساختارهای اطلاعات احتمالی
- بررسی صحت در پروتکل‌های تعاملی

نحوه استفاده:

```
int crypto_shorthash(unsigned char *out, const unsigned char *in,
                     unsigned long long inlen, const unsigned char *k);
```

این تابع یک اثرانگشت با طول ثابت (از نوع بایت‌های crypto_shorthash_BYTES) برای پیام in به طول inlen بایت و با استفاده از کلید k محاسبه می‌کند. k از نوع بایت‌های crypto_shorthash_KEYBYTES می‌باشد و با استفاده از تابع randombytes_buf() ساخته می‌شود.

پیام‌های مشابهی که با استفاده از کلیدهای یکسان از آن‌ها hash گرفته می‌شود، همیشه دارای نتایج خروجی یکسان می‌باشند.

ثابت‌ها:

- crypto_shorthash_BYTES
- crypto_shorthash_KEYBYTES

جزئیات الگوریتم:

- SipHash-2-4

۱۰-۲ Password Hashing

کلیدهای خصوصی که برای رمزنگاری و یا امضای اطلاعات محرمانه استفاده می‌شوند، باید از یک فضای بسیار بزرگ کلیدها انتخاب شوند. به هر حال، رمزهای عبور معمولاً کوتاه هستند و از رشته‌های ساخته افراد به‌وجود می‌آیند. به همین دلیل حمله دیکشنری می‌تواند بر روی آن‌ها کارساز شود.

توابع مربوط به password hashing، نیازمند مقادیر کلیدخصوصی، رمز عبور (با اندازه دلخواه) و یک مقدار salt بوده و دارای ویژگی زیر هستند:

- طول کلید تولید شده توسط برنامه تعیین می‌شود و به طول رمز عبور بستگی ندارد.
- رمزهای عبوری که با پارامترهای یکسان از آن‌ها hash گرفته می‌شود، دارای خروجی یکسان خواهند بود.
- رمزهای عبوری که با salt های گوناگون از آن‌ها hash گرفته می‌شود، خروجی‌های متفاوت تولید می‌کنند.
- توابعی که یک کلید از یک رمز عبور و یک salt مشتق می‌گیرند، نیاز به فعالیت زیاد CPU و حافظه کافی دارند. بنابراین، با توجه به تلاش قابل توجه برای تصدیق هر رمزعبور، اثر حملات brute-force کاهش می‌یابد.

crypto_pwhash_* که یک API سطح بالای مربوط به sodium است، از توابع مربوط به Argon2 استفاده می‌کند.

API های خاص crypto_pwhash_scryptsalsa208sha256_* از توابع محافظه‌کارتر و گسترده‌تر Scrypt استفاده می‌کنند.

Argon2 :

Argon2 دارای دو نسخه می‌باشد: Argon2d و Argon2i. نحوه دسترسی به حافظه در Argon2i مستقل از اطلاعات است که برای password hashing و اشتقاق کلید بر اساس رمزهای عبور ترجیح داده می‌شود. همچنین Argon2i برای محافظت در برابر حملات tradeoff، چندین راه در طول حافظه ایجاد می‌کند. این قابلیت در sodium از نسخه 1.0.9 به بعد پیاده‌سازی شده است. اگر استفاده از sodium 1.0.9 و نسخه‌های بعدی آن میسر باشد، توصیه می‌شود که از Argon2 به‌جای Scrypt استفاده شود.

Scrypt :

Scrypt بدین منظور طراحی شده است که انجام حملات سخت‌افزاری در مقیاس بزرگ به دلیل نیاز به میزان زیادی از حافظه، برای حمله‌کننده پر هزینه باشد. اگرچه میزان حافظه می‌تواند به طور قابل توجهی هزینه محاسبات اضافی را کاهش دهد، ولی این تابع امروزه همچنان یک انتخاب عالی محسوب می‌شود. Scrypt در sodium از نسخه 0.5.0 به بعد پیاده‌سازی شده است. اگر سازگاری با نسخه‌های پایین‌تر مدنظر باشد، استفاده از Scrypt نسبت به Argon2 توصیه می‌شود.

۲-۱۰-۱ تابع Argon2

از نسخه sodium 1.0.9، یک طرح جدید برای passrord hashing به نام Argon2 به آن اضافه شده است. Argon2 با بالاترین نرخ حافظه را پر می‌کند و از واحدهای محاسباتی بهترین استفاده را می‌برد. البته همچنان در برابر حملات tradeoff مقاوم است.

مثال: اشتقاق کلید

```
#define PASSWORD "Correct Horse Battery Staple"
#define KEY_LEN crypto_box_SEEDBYTES

unsigned char salt[crypto_pwhash_SALTBYTES];
unsigned char key[KEY_LEN];

randombytes_buf(salt, sizeof salt);
```

```
if (crypto_pwhash
    (key, sizeof key, PASSWORD, strlen(PASSWORD), salt,
     crypto_pwhash_OPSLIMIT_INTERACTIVE,
crypto_pwhash_MEMLIMIT_INTERACTIVE,
     crypto_pwhash_ALG_DEFAULT) != 0) {
    /* out of memory */
}
```

مثال: ذخیره رمز عبور

```
#define PASSWORD "Correct Horse Battery Staple"

char hashed_password[crypto_pwhash_STRBYTES];

if (crypto_pwhash_str
    (hashed_password, PASSWORD, strlen(PASSWORD),
     crypto_pwhash_OPSLIMIT_SENSITIVE,  crypto_pwhash_MEMLIMIT_SENSITIVE)  !=  0)
{
    /* out of memory */
}

if (crypto_pwhash_str_verify
    (hashed_password, PASSWORD, strlen(PASSWORD)) != 0) {
    /* wrong password */
}
```

اشتقاق کلید:

```
int crypto_pwhash(unsigned char * const out,
                  unsigned long long outlen,
                  const char * const passwd,
                  unsigned long long passwdlen,
                  const unsigned char * const salt,
                  unsigned long long opslimit,
                  size_t memlimit, int alg);
```

تابع ()crypto_pwhash یک کلید با طول outlen بایت از رمزعبور passwd به طول passwdlen و salt موجود در متغیر salt که دارای طول ثابت crypto_pwhash_SALTBYTES می‌باشد، تولید می‌کند. طول outlen باید حداقل 16 بایت (128 بیت) باشد. کلید محاسبه شده در out ذخیره می‌شود.

opslimit نشان‌دهنده حداکثر میزان محاسبات برای انجام وظیفه مورد نظر می‌باشد. افزایش این عدد مبین نیاز به پردازش بیشتر CPU برای محاسبه کلید می‌باشد.

memlimit حداکثر میزان نیاز به RAM بر حسب بایت می‌باشد.

alg یک مشخصه از الگوریتم مورد استفاده می‌باشد و باید با حالت crypto_pwhash_ALG_DEFAULT مقداردهی شود.

برای عملیات‌های آنلاین و تعاملی از پارامترهای crypto_pwhash_OPSLIMIT_INTERACTIVE و crypto_pwhash_MEMLIMIT_INTERACTIVE استفاده می‌شود که به 32 Mb از RAM نیاز دارد. مقادیر بالاتر می‌تواند به افزایش امنیت کمک کند.

همچنین crypto_pwhash_OPSLIMIT_MODERATE و crypto_pwhash_MEMLIMIT_MODERATE می‌توانند استفاده شوند. این پارامترها نیاز به 128 Mb فضا از حافظه RAM دارند. با استفاده از یک CPU Core i7 با فرکانس 2.8 GHz ، زمانی در حدود 0.7 ثانیه صرف تولید کلید می‌شود.

برای اطلاعات با حساسیت بالا و غیرتعاملی، پارامترهای crypto_pwhash_OPSLIMIT_SENSITIVE و crypto_pwhash_MEMLIMIT_SENSITIVE استفاده می‌شوند. با این پارامترها، استخراج کلید با استفاده از

یک CPU Core i7 با فرکانس 2.8 GHz در حدود 3.5 ثانیه طول می‌کشد و نیاز به 512 Mb حافظه تخصیص داده شده RAM دارد.

salt باید غیرقابل پیش‌بینی باشد. استفاده از تابع randombytes_buf() آسانترین روش برای پر کردن بایت‌های crypto_pwhash_SALTBYTES با salt می‌باشد.

البته بایستی به این نکته دقت شود که اگر هدف، تولید کلید یکسان از رمزهای عبور یکسان باشد، باید از مقادیر مشابه salt، opslimit و memlimit استفاده شود. بنابراین این پارامترها باید برای هر کاربر ذخیره شوند.

این تابع در صورت موفقیت‌آمیز بودن عدد 0 و در غیر این صورت و ناتمام ماندن محاسبه کلید عدد 1- را باز می‌گرداند.

ذخیره‌سازی رمز عبور:

```
int crypto_pwhash_str(char out[crypto_pwhash_STRBYTES],
                      const char * const passwd,
                      unsigned long long passwdlen,
                      unsigned long long opslimit,
                      size_t memlimit);
```

تابع crypto_pwhash_str() یک رشته اسکی کد شده را در out ذخیره می‌کند که شامل موارد زیر است:

- نتیجه مربوط به تابع hash اعمال شده بر روی رمز عبور passwd با طول passwdlen
- Salt تولید شده به صورت خودکار برای محاسبات قبلی
- پارامترهای دیگری که برای بررسی و تصدیق رمز عبور مورد نیاز است (شامل مشخصه الگوریتم، نسخه آن، opslimit و memlimit).

پارامتر out به منظور ذخیره بایت‌های crypto_pwhash_STRBYTES باید به اندازه کافی بزرگ باشد، اما رشته واقعی خروجی ممکن است کوتاه‌تر باشد. رشته خروجی با 0 تمام می‌شود و فقط شامل کاراکترهای

ASCII می‌باشد که می‌تواند به راحتی در دیتابیس‌های SQL و یا دیتابیس‌های دیگر ذخیره شود. اطلاعات اضافه دیگری به منظور بررسی و تصدیق رمز عبور ذخیره نمی‌شود.

اگر تابع به صورت موفقیت‌آمیز به اتمام برسد، عدد 0 و در غیر این صورت عدد 1- را باز می‌گرداند.

```
int crypto_pwhash_str_verify(const char str[crypto_pwhash_STRBYTES],
                             const char * const passwd,
                             unsigned long long passwdlen);
```

این تابع اعتبار رمز عبور str را بدین صورت که آیا یک رشته احراز اصالت معتبر (که توسط تابع crypto_pwhash_str() تولید شده است) برای passwd به طول passwdlen می‌باشد یا خیر، بررسی و تصدیق می‌کند. اگر نتیجه بررسی موفقیت‌آمیز باشد، تابع عدد 0 و در غیر این صورت عدد 1- را باز می‌گرداند.

ثابت‌ها:

- crypto_pwhash_ALG_DEFAULT
- crypto_pwhash_SALTBYTES
- crypto_pwhash_STRBYTES
- crypto_pwhash_STRPREFIX
- crypto_pwhash_OPSLIMIT_INTERACTIVE
- crypto_pwhash_MEMLIMIT_INTERACTIVE
- crypto_pwhash_OPSLIMIT_MODERATE
- crypto_pwhash_MEMLIMIT_MODERATE
- crypto_pwhash_OPSLIMIT_SENSITIVE
- crypto_pwhash_MEMLIMIT_SENSITIVE

۲-۱۰-۲ تابع Scrypt

مثال: اشتقاق کلید

```
#define PASSWORD "Correct Horse Battery Staple"
#define KEY_LEN crypto_box_SEEDBYTES
```

```
unsigned char salt[crypto_pwhash_scryptsalsa208sha256_SALTBYTES];
unsigned char key[KEY_LEN];

randombytes_buf(salt, sizeof salt);

if (crypto_pwhash_scryptsalsa208sha256
   (key, sizeof key, PASSWORD, strlen(PASSWORD), salt,
    crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE,
    crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE) != 0) {
   /* out of memory */
}
```

مثال: ذخیره‌سازی رمزعبور

```
#define PASSWORD "Correct Horse Battery Staple"

char hashed_password[crypto_pwhash_scryptsalsa208sha256_STRBYTES];

if (crypto_pwhash_scryptsalsa208sha256_str
   (hashed_password, PASSWORD, strlen(PASSWORD),
    crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_SENSITIVE,
    crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_SENSITIVE) != 0) {
   /* out of memory */
}

if (crypto_pwhash_scryptsalsa208sha256_str_verify
   (hashed_password, PASSWORD, strlen(PASSWORD)) != 0) {
   /* wrong password */
}
```

اشتقاق کلید:

```
int crypto_pwhash_scryptsalsa208sha256(unsigned char * const out,
                                       unsigned long long outlen,
                                       const char * const passwd,
                                       unsigned long long passwdlen,
                                       const unsigned char * const salt,
```

```
unsigned long long opslimit,
size_t memlimit);
```

تابع () crypto_pwhash_scryptsalsa208sha256 یک کلید با طول outlen بایت از رمز عبور passwd به طول passwdlen و salt موجود در متغیر salt که دارای طول ثابت crypto_pwhash_scryptsalsa208sha256_SALTBYTES می‌باشد، مشتق می‌گیرد. کلید محاسبه شده در out ذخیره می‌شود. opslimit نشان‌دهنده حداکثر میزان محاسبات برای انجام وظیفه مورد نظر می‌باشد. افزایش این عدد مبین نیاز به پردازش بیشتر CPU برای محاسبه کلید می‌باشد.

memlimit حداکثر میزان نیاز به RAM بر حسب بایت می‌باشد. اکیداً توصیه می‌شود که حداقل 16 مگابایت در اختیار برنامه قرار داده شود.

برای عملیات‌های آنلاین و تعاملی از پارامترهای crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE و crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE استفاده می‌شود. مقادیر بالاتر می‌تواند به افزایش امنیت کمک کند.

برای اطلاعات با حساسیت بالا، پارامترهای rypto_pwhash_scryptsalsa208sha256_OPSLIMIT_SENSITIVE و crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_SENSITIVE استفاده می‌شوند. با این پارامترها، استخراج کلید با استفاده از یک CPU Core i7 با فرکانس 2.8 GHz در حدود 2 ثانیه طول می‌کشد و نیاز به بیش از 1 Gb حافظه تخصیص داده شده RAM دارد.

باید مقدار salt غیرقابل پیش‌بینی باشد. استفاده از تابع randombytes_buf() آسان‌ترین روش برای پر کردن بایت‌های crypto_pwhash_scryptsalsa208sha256_SALTBYTES مربوط به salt می‌باشد. البته بایستی به

این نکته دقت شود که اگر هدف، تولید کلید یکسان از رمزهای عبور یکسان باشد، باید از مقادیر مشابه salt، opslimit و memlimit استفاده شود. بنابراین این پارامترها باید برای هر کاربر ذخیره شوند.

این تابع در صورت موفقیت‌آمیز بودن عدد 0 و در غیر این صورت و ناتمام ماندن محاسبه کلید عدد 1- را باز می‌گرداند.

ذخیره‌سازی رمزعبور:

```
int crypto_pwhash_scryptsalsa208sha256_str (char
out[crypto_pwhash_scryptsalsa208sha256_STRBYTES],
                const char * const passwd,
                unsigned long long passwdlen,
                unsigned long long opslimit,
                size_t memlimit);
```

تابع crypto_pwhash_scryptsalsa208sha256_str () یک رشته ASCII کد شده را در out ذخیره می‌کند که شامل موارد زیر است:

- نتیجه مربوط به تابع hash اعمال شده بر روی رمزعبور passwd با طول passwdlen
- Salt تولید شده به صورت خودکار برای محاسبات قبلی
- پارامترهای دیگری که برای بررسی و تصدیق رمز عبور مورد نیاز است (شامل مشخصه الگوریتم، نسخه آن، opslimit و memlimit).

پارامترهای crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE و crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE مقادیر پایه به منظور استفاده برای opslimit و memlimit می‌باشند.

رشته خروجی با 0 تمام می‌شود و فقط شامل کاراکترهای ASCII می‌باشد که می‌تواند به راحتی در دیتابیس‌های SQL و یا دیتابیس‌های دیگر ذخیره شود. اطلاعات اضافه دیگری به منظور بررسی و تصدیق رمز عبور ذخیره نمی‌شود.

اگر تابع به صورت موفقیت‌آمیز به اتمام برسد، عدد 0 و در غیر این صورت عدد 1- را باز می‌گرداند.

int crypto_pwhash_scryptsalsa208sha256_str_verify(const char str[crypto_pwhash_scryptsalsa208sha256_STRBYTES,[
const char * const passwd,
unsigned long long passwdlen;

این تابع اعتبار رمز عبور str را بدین صورت که آیا یک رشته احراز اصالت معتبر (که توسط تابع crypto_pwhash_scryptsalsa208sha256_str () تولید شده است) برای passwd به طول passwdlen می‌باشد یا خیر، بررسی و تصدیق می‌کند. همچنین str با 0 خاتمه می‌یابد.

اگر نتیجه بررسی موفقیت‌آمیز باشد، تابع عدد 0 و در غیر این صورت عدد 1- را باز می‌گرداند.

ثابت‌ها:

- crypto_pwhash_scryptsalsa208sha256_SALTBYTES
- crypto_pwhash_scryptsalsa208sha256_STRBYTES
- crypto_pwhash_scryptsalsa208sha256_STRPREFIX
- crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE
- crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE
- crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_SENSITIVE
- crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_SENSITIVE

۱۱-۲ تبادل کلید

Sodium از X25519 به منظور استفاده از توابع Diffie-Hellman بهره می‌گیرد.

نحوه استفاده:

int crypto_scalarmult_base(unsigned char *q, const unsigned char *n);

با توجه به کلید خصوصی n مربوط به کاربر (از نوع بایت‌های crypto_scalarmult_SCALARBYTES)، تابع crypto_scalarmult_base() کلیدعمومی کاربر را محاسبه و در q (از نوع بایت‌های crypto_scalarmult_BYTES) ذخیره می‌کند.

```
int crypto_scalarmult(unsigned char *q, const unsigned char *n,
                      const unsigned char *p);
```

این تابع برای محاسبه کلید رمز مشترک q با توجه به کلیدخصوصی کاربر و کلیدعمومی کاربران دیگر استفاده می‌شود. n دارای طول crypto_scalarmult_SCALARBYTES بایت و p و خروجی دارای طول crypto_scalarmult_BYTES می‌باشند.

```
unsigned char client_publickey[crypto_box_PUBLICKEYBYTES];
unsigned char client_secretkey[crypto_box_SECRETKEYBYTES];
unsigned char server_publickey[crypto_box_PUBLICKEYBYTES];
unsigned char server_secretkey[crypto_box_SECRETKEYBYTES];
unsigned char scalarmult_q_by_client[crypto_scalarmult_BYTES];
unsigned char scalarmult_q_by_server[crypto_scalarmult_BYTES];
unsigned char sharedkey_by_client[crypto_generichash_BYTES];
unsigned char sharedkey_by_server[crypto_generichash_BYTES];
crypto_generichash_state h;

/* Create client's secret and public keys */
randombytes_buf(client_secretkey, sizeof client_secretkey);
crypto_scalarmult_base(client_publickey, client_secretkey);

/* Create server's secret and public keys */
randombytes_buf(server_secretkey, sizeof server_secretkey);
crypto_scalarmult_base(server_publickey, server_secretkey);

/* The client derives a shared key from its secret key and the server's public key */
/* shared key = h(q || client_publickey || server_publickey) */
if (crypto_scalarmult(scalarmult_q_by_client, client_secretkey, server_publickey) != 0) {
   /* Error */
}
```

```
crypto_generichash_init(&h, NULL, 0U, crypto_generichash_BYTES);
crypto_generichash_update(&h, scalarmult_q_by_client, sizeof scalarmult_q_by_client);
crypto_generichash_update(&h, client_publickey, sizeof client_publickey);
crypto_generichash_update(&h, server_publickey, sizeof server_publickey);
crypto_generichash_final(&h, sharedkey_by_client, sizeof sharedkey_by_client);

/* The server derives a shared key from its secret key and the client's public key */
/* shared key = h(q || client_publickey || server_publickey) */
if (crypto_scalarmult(scalarmult_q_by_server, server_secretkey, client_publickey) != 0) {
   /* Error */
}
crypto_generichash_init(&h, NULL, 0U, crypto_generichash_BYTES);
crypto_generichash_update(&h, scalarmult_q_by_server, sizeof scalarmult_q_by_server);
crypto_generichash_update(&h, client_publickey, sizeof client_publickey);
crypto_generichash_update(&h, server_publickey, sizeof server_publickey);
crypto_generichash_final(&h, sharedkey_by_server, sizeof sharedkey_by_server);

/* sharedkey_by_client and sharedkey_by_server are identical */
```

ثابت‌ها:

- crypto_scalarmult_BYTES
- crypto_scalarmult_SCALARBYTES

۱۲-۲ اشتقاق کلید

اشتقاق کلید از رمز عبور: عملیات pwhash یک کلیدخصوصی به هر اندازه‌ای از یک رمز عبور و salt مشتق می-گیرد.

اشتقاق چندین کلید از یک کلید: می‌توان چندین کلیدخصوصی از یک کلید اصلی مشتق گرفت.

با توجه به یک کلید اصلی و مشخصه آن، می‌توان به طور قطعی یک زیرکلید محاسبه کرد. البته حمله‌کننده با داشتن این زیرکلید نمی‌تواند کلید اصلی و یا دیگر زیرکلیدها را محاسبه کند. برای انجام این کار، از تابع Blake2 که یک تابع hash می‌باشد به جای ساختار HKDF استفاده می‌شود:

```
const unsigned char appid[crypto_generichash_blake2b_PERSONALBYTES] = {
   'A', ' ', 'S', 'i', 'm', 'p', 'l', 'e', ' ', 'E', 'x', 'a', 'm', 'p', 'l', 'e'
};
unsigned char keyid[crypto_generichash_blake2b_SALTBYTES] = {0};
unsigned char masterkey[64];
unsigned char subkey1[16];
unsigned char subkey2[32];

/* Generate a master key */
randombytes_buf(masterkey, sizeof masterkey);

/* Derive a first subkey (id=0) */
crypto_generichash_blake2b_salt_personal(subkey1, sizeof subkey1,
                                         NULL, 0,
                                         masterkey, sizeof masterkey,
                                         keyid, appid);

/* Derive a second subkey (id=1) */
sodium_increment(keyid, sizeof keyid);
crypto_generichash_blake2b_salt_personal(subkey2, sizeof subkey2,
                                         NULL, 0,
                                         masterkey, sizeof masterkey,
                                         keyid, appid);
```

تابع crypto_generichash_blake2b_salt_personal() می‌تواند برای اشتقاق یک زیرکلید با هر اندازه‌ای از یک کلید استفاده شود. در این مثال، دو زیرکلید از یک کلید اصلی مشتق گرفته می‌شود. این زیرکلیدها دارای اندازه‌های متفاوت هستند (128 بیت و 256 بیت) و از یک کلید اصلی به طول 512 بیت مشتق شده‌اند.

پارامتر appid یک مقدار 16 بیتی و فاش می‌باشد. با استفاده از این مقدار، یک زوج (masterkey, keyid) یکسان در برنامه‌های مختلف، خروجی متفاوت تولید می‌کنند.

همچنین keyid که یک salt می‌باشد، نیازی به محرمانه بودن ندارد و فاش می‌باشد. Keyid یک مقدار 16 بیتی است و می‌تواند یک شمارشگر ساده باشد و برای اشتقاق بیشتر از یک کلید از کلید اصلی استفاده می‌شود.

اضافه کردن نانس:

```
int crypto_core_hchacha20(unsigned char *out, const unsigned char *in,
                          const unsigned char *k, const unsigned char *c);
```

این تابع یک کلیدخصوصی k به طول 32 بایت (از نوع بایت‌های crypto_core_hchacha20_KEYBYTES) و متغیر in به طول 16 بایت (از نوع بایت‌های crypto_core_hchacha20_INPUTBYTES) را به عنوان ورودی دریافت می‌کند و یک مقدار 32 بایتی (ازنوع بایت‌های crypto_core_hchacha20_OUTPUTBYTES) غیر قابل تشخیص از اعداد تصادفی بدون دانستن k به عنوان خروجی تولید می‌کند.

کد زیر در ساختار ChaCha20-Poly1305 به همراه یک نانس 192 بیتی استفاده می‌شود:

```
#define MESSAGE (const unsigned char *) "message"
#define MESSAGE_LEN 7

unsigned char c[crypto_aead_chacha20poly1305_ABYTES + MESSAGE_LEN];
unsigned char k[crypto_core_hchacha20_KEYBYTES];
unsigned char k2[crypto_core_hchacha20_OUTPUTBYTES];
unsigned char n[crypto_core_hchacha20_INPUTBYTES +
                crypto_aead_chacha20poly1305_NPUBBYTES];

randombytes_buf(k, sizeof k);
randombytes_buf(n, sizeof n); /* 192-bits nonce */

crypto_core_hchacha20(k2, n, k, NULL);

assert(crypto_aead_chacha20poly1305_KEYBYTES <= sizeof k2);
assert(crypto_aead_chacha20poly1305_NPUBBYTES ==
```

```
    (sizeof n) - crypto_core_hchacha20_INPUTBYTES);

crypto_aead_chacha20poly1305_encrypt(c, NULL, MESSAGE, MESSAGE_LEN,
                                     NULL, 0, NULL,
                                     n + crypto_core_hchacha20_INPUTBYTES,
                                     k2);
```

۳ نحوه ذخیره امن رمزهای عبور در زبان‌های مختلف بدون استفاده از libsodium

اگر به هر دلیل امکان استفاده از libsodium وجود نداشته باشد، از کتابخانه‌های رمزنگاری مختلف در هر کدام از زبان‌ها استفاده می‌شود. در این بخش نحوه ذخیره امن رمزهای عبور به همراه مثال‌های در این خصوص در زبان های برنامه‌نویسی مختلف بررسی می‌شوند.

۳-۱ الگوریتم‌های قابل قبول درهم‌سازی

در مورد رتبه‌بندی این الگوریتم‌ها اختلاف نظر وجود دارد، ولی افراد حرفه‌ای در زمینه رمزنگاری معتقدند که در سال 2016 باید از الگوریتم‌های زیر به منظور ذخیره امن رمزهای عبور استفاده کرد:

- Argon2، برنده مسابقات password hashing
- bcrypt
- scrypt
- Catena
- Lyra2
- Makwa
- yescrypt
- PBKDF2

۲-۳ ذخیره امن رمزهای عبور در زبان PHH

ابتدا باید بررسی شود که از نسخه‌های پشتیبانی شده PHP استفاده می‌شود. برای مشاهده این نسخه‌ها می‌توان به آدرس زیر مراجعه کرد:

https://secure.php.net/supported-versions.php

اگر از یک نسخه پشتیبانی شده استفاده می‌شود، بنابراین password API های زبان PHP قابل استفاده می‌باشند. توابع مربوط به این عملیات در آدرس زیر قابل مشاهده هستند:

https://secure.php.net/manual/en/ref.password.php

اگر از نسخه‌های پشتیبانی‌شده استفاده نمی‌شود، باید حتماً آن را ارتقاء داد. در صورتی که امکان ارتقای نسخه فعلی وجود نداشته باشد می‌توان از کتابخانه password_compat استفاده کرد. نحوه استفاده از این کتابخانه در آدرس زیر توضیح داده شده است:

https://github.com/ircmaxell/password_compat

```
$hash = password_hash($userPassword, PASSWORD_DEFAULT);
```

این تابع با استفاده از یک الگوریتم قدرتمند، hash مربوط به رمزعبور مورد نظر را تولید می‌کند. کمترین میزان cost در نظر گرفته شده باید برابر 10 باشد. البته اگر سخت‌افزار مورد نظر از اعداد بزرگتر نیز پشتیبانی می‌کند، می‌توان مقدار 12 را در نظر گرفت. به هر حال مقدار پیش‌فرض برای cost برابر 10 می‌باشد.

```
$hash = password_hash($userPassword, PASSWORD_DEFAULT, ['cost' => 12]);
```

نحوه تصدیق یک رمزعبور در برابر یک مقدار hash به روش زیر انجام می‌شود:

```
if (password_verify($userPassword, $hash)) {
   // Login successful.
   if (password_needs_rehash($hash, PASSWORD_DEFAULT, ['cost' => 12])) {
      // Recalculate a new password_hash() and overwrite the one we stored previously
   }
}
```

تا نسخه 7 از PHP، PASSWORD_DEFAULT همچنان از الگوریتم bcrypt استفاده می‌کند. در نسخه‌های بعدی احتمال جایگزینی آن با الگوریتم Argon2 وجود دارد.

Password hashing در زبان PHP با استفاده از الگوریتم Scrypt :

```
# If you don't have PECL installed, get that first.
pecl install scrypt
echo "extension=scrypt.so" > /etc/php5/mods-available/scrypt.ini
php5enmod scrypt
```

در ادامه یک کپی از کد موجود در آدرس زیر باید به پروژه اضافه کرد.

```
https://github.com/DomBlack/php-scrypt/blob/master/scrypt.php
# Hashing
$hash = \Password::hash($userProvidedPassword);
# Validation
if (\Password::check($userProvidedPassword, $hash)) {
   // Logged in successfully.
}
```

۳-۳ ذخیره امن رمزهای عبور در زبان Java

بهترین روش برای ذخیره امن رمزهای عبور در زبان java به غیر از استفاده از libsodium، استفاده از الگوریتم jBCrypt است که همان الگوریتم bcrypt را برای زبان java فراهم می‌کند.

```
String hash = BCrypt.hashpw(userProvidedPassword, BCrypt.gensalt());
```

نحوه تصدیق یک مقدار hash تولیدشده توسط الگوریتم bcrypt :

```
if (BCrypt.checkpw(userProvidedPassword, hash)) {
   // Login successful.
}
```

استفاده از الگوریتم Scrypt در زبان Java:

```
# Calculating a hash
int N = 16384;
```

```
int r = 8;
int p = 1;
String hashed = SCryptUtil.scrypt(passwd, N, r, p);

# Validating a hash
if (SCryptUtil.check(passwd, hashed)) {
   // Login successful
}
```

۴-۳ ذخیره امن رمزهای عبور در زبان C# (.NET)

برای این منظور استفاده از BCrypt.NET به همراه System.Security.Cryptography.Rfc2898DeriveBytes نسبت به PBKDF2-SHA1 توصیه می‌شود.

```
// Calculating a hash
string hash = BCrypt.HashPassword(usersPassword, BCrypt.GenerateSalt());

// Validating a hash
if (BCrypt.Verify(usersPassword, hash)) {
   // Login successful
}
```

استفاده از Scrypt در زبان C#

```
// This is necessary:
ScryptEncoder encoder = new ScryptEncoder();
// Calculating a hash
SecureString hashedPassword = encoder.Encode(usersPassword);
// Validating a hash
if (encoder.Compare(usersPassword, hashedPassword)) {
   // Login successful
}
```

۳-۵ ذخیره امن رمزهای عبور در زبان Ruby

```
require "bcrypt"

# Calculating a hash
my_password = BCrypt::Password.create(usersPassword)
# Validating a hash
if my_password == usersPassword
  # Login successful
```

استفاده از Scrypt در زبان Ruby

```
require "scrypt"

# Calculating a hash
password = SCrypt::Password.create(usersPassword)
# Validating a hash
if password == usersPassword
  # Login successful
```

۳-۶ ذخیره امن رمزهای عبور در زبان Python

```
import bcrypt
import hmac

# Calculating a hash
password = b"correct horse battery staple"
hashed = bcrypt.hashpw(password, bcrypt.gensalt())
# Validating a hash (don't use ==)
if (hmac.compare_digest(bcrypt.hashpw(password, hashed), hashed)):
    # Login successful
```

همچنین در زبان python می‌توان از کتابخانه passlib استفاده کرد:

```
from passlib.hash import bcrypt

# Calculating a hash
hash = bcrypt.encrypt(usersPassword, rounds=12)
```

```
# Validating a hash
if bcrypt.verify(usersPassword, hash):
    # Login successful
```

۳-۷ ذخیره امن رمزهای عبور در زبان Node.js

دو پیاده‌سازی امنیتی مختلف از bcrypt در زبان Node.js وجود دارد:

```
var Promise = require("bluebird");
var bcrypt = Promise.promisifyAll(require("bcrypt"));

function addBcryptType(err) {
    // Compensate for `bcrypt` not using identifiable error types
    err.type = "bcryptError";
    throw err;
}

// Calculating a hash:
Promise.try(function() {
    return bcrypt.hashAsync(usersPassword, 10).catch(addBcryptType);
}).then(function(hash) {
    // Store hash in your password DB.
});

// Validating a hash:
// Load hash from your password DB.
Promise.try(function() {
    return bcrypt.compareAsync(usersPassword, hash).catch(addBcryptType);
}).then(function(valid) {
    if (valid) {
        // Login successful
    } else {
        // Login wrong
    }
});
```

```
// You would handle errors something like this, but only at the top-most point where it makes sense to do so:
Promise.try(function() {
   // Generate or compare a hash here
}).then(function(result) {
   // ... some other stuff ...
}).catch({type: "bcryptError"}, function(err) {
   // Something went wrong with bcrypt
});
```

استفاده از Scrypt در زبان Node.js

```
var Promise = require("bluebird");
var scrypt = require("scrypt-for-humans");

// Calculating a hash:
Promise.try(function() {
  return scrypt.hash(usersPassword);
}).then(function(hash) {
  // Store hash for long term use
});

// Validating a hash:
Promise.try(function() {
  return scrypt.verifyHash(usersPassword, hash);
}).then(function() {
  // Login successful
}).catch(scrypt.PasswordError, function(err) {
  // Login failed
});
```

پیوست ۱:

کلید RSA

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1 (OpenBSD)

mQINBFTZ0A8BEAD2/BeYhJpEJDADNuOz5EO8E0SIj5VeQdb9WLh6tBe37KrJJy7+
FBFnsd/ahfsqoLmr/IUE3+ZejNJ6QVozUKUAbds1LnKh8ejX/QegMrtgb+F2Zs83
8ju4k6GtWquW5OmiG7+b5t8R/oHlPs/1nHbk7jkQqLkYAYswRmKld1rqrrLFV8fH
SAsnTkgeNxpX8W4MJR22yEwxb/k9grQTxnKHHkjJInoP6VnGRR+wmXL/7xeyUg6r
EVmTaqEoZA2LiSaxaJ1c8+5c7oJ3zSBUveJA587KsCp56xUKcwm2IFJnC34WiBDn
KOLB7lNxIT3BnnzabF2m+5602qWRbyMME2YZmcISQzjiVKt8O62qmKfFr5u9B8Tx
iYpSOal9HvZqih8C7u/SKeGzbONfbmmJgFuA15LVwt7I5Xx7565+kWeoDgKPlfrL
7zPrCQqS1a75MB+W/fOHhCRJ3IqFc+dT1F4hb8AAKWrERVq27LEJzmxXH36kMbB+
eQg336JlS6TmqelVFb15PgtcFh972jJK8u/vpHY0EBPij5chjYQ2nCBmFLT5O4UZ
Y4Gm8Z3QLFG1EeOiz+uRdNfchxwfLkjng1UhKXSq5yuOAAeMaNoYFtCf1hAHG6tx
vWyIijRxUd5c8cDZsKMuLQ34O6DuvPZyeCy6q8BTfW18miMMhIH0QTS9MwARAQAB
tC5GcmFuayBEZW5pcyAoSmVkaS9TZWN0b3IgT25lKSA8akBwdXJlZnRwZC5vcmc+
iQI2BBMBCAAgAhsDAh4BAheABQJU2dF6BAsJCAcFFQoJCAsFFgIDAQAACgkQIQYn
qrpwn+GpOBAAkJu5yZhLPBIznDZMr0oJ/pJiSea7GUCY4fVuFUKLpLlSjIaSxC4E
2oWG8cJoMdMhwW1x166rRZPdXFpW8eC5r+h8m25HBJ649FjMUPDi2r9uQgPdBy80
I+gFlrsinSy7xbdlUSpjrcYYCx9jYjjTwH6L1QZa+YCMFya8dob/NcdzQ0o7cNRu
5NG988cScsscXYXzI6SMouSwPGCMrQHAsM31Yb8YFbJLuDxFRCZY5+qiR8DXDzW4
Lp68fJq0X/UGW9Q+i29LMTvZZWDGBQ9bwQNtvDrPZ8SYp249cMOsR4W7FK4Y0Oea
YRTBFcXaeXEKAP1ZqYrY22BDiHJO5IGY72D3j3vPATAYigwjr/qNFOt/DaERFpQ4
L7RD+E6WLHATFWxZHH/APck6q8bY4EHr8GJWA77sIqN/Ctvap759QKB8nrerT6lA
0cojhS5Ie8Lro6YsMAXDqwjzsv+VgnTgql8oAFmuU+o+6cmHUwGNHgEs+xe2UDQi
kxu685gOCHfHmBwue391glHufQdveChy5eikif6q6Ndg7VH9mR335o8VJ9I+Vp/k
3W8XZBA9OEuwrxjy1EzWvcb2WGXrUHVZ32w+E9CICvFFV7JiTntG3t1Ch4/bbFwr
wdkc5EZTh0c6B7YfIkEWnOnBovWBPEBkSGve371MsqBuKuBr1W4jecyIRgQQEQgA
BgUCVNnRHAAKCRCSa8UXHN6kOWXzAKCGlk6DvVCqExkBd6OEsaEoOBgH5ACfcVQa
z/FEgCdRsJeLi7xNwZXZ22O0IUZyYW5rIERlbmlzIDxnaXRodWJAcHVyZWZ0cGQu
b3JnPokCNgQTAQgAIAIbAwIeAQIXgAUCVNnRaQQLCQgHBRUKCQgLBRYCAwEAAAoJ
ECEGJ6q6cJ/hslIQAI2l+uRlwmofiSHo/H2cUDNO2Nn7uRfcVIw9EItTmdU6KKx9
nkgFP3Y3lUwkLQFP6aQhQJyHBU5QGqn9n8k8+jEPciTL7hcbTuY0YRuz0mp9bJ8r
ruqGxTrZuogvIVntwnn1HvgAbu13HKu+3KOLYDmWqosVNf0a8GjHj10ZDuNDPQVb
X6NWDes+jLdeUsxVKUZHlOC3CiRCSHJzZ3G1gO9QU78LQAFCIIDO7GO7xPjqbvEX
nsys5f12OLXB4NqBlIamEdyztV+CwIZBM9Ni6ytPnEhWzTHzHwi95oNa+AtpUlgG
RYjYtMR9pxCqVkrplwrwhA4dbSO7HLiXQIrA57F1/5LwKRR4e7IGhnTpZoW8hr8y
qg4AAVCZqr5aB82LOJAMP6ZlC7kQb9/YxGYw4Vwf6qCY8Iw74MvIL5wW0zSv/orB
eNtHeP0Z/Ozx3UXKA2chNElEWbZ9e0IZBXgcj/JDfK8e0VTqv1ItHLm2ZkvCbyhV
fER8I8AHPnfzwkXvWFeDKeMO8rakqDeNQ3h4BeiCBCVHpEsUdIWSG3oCO1guy9/h
xMJR2yAWiK+35sCcZbrgTTN0oQepRMuZ34niIBK0jUh7t1M5sBMNgxEAIeKjJf64
DEudNz+xUgek5N+BXx7hryuVC3s1y6H42ztOjPtpHPVUw98gWpv5V7QRLBs0iEYE

EBEIAAYFAlTZ0RwACgkQkmvFFxzepDn8sACdF51BycwRvMpkFPea1Yi3/B1EOs0A
oJT9afe3zQnOlcIuBFBzpdOTsecUtCZGcmFuayBEZW5pcyA8ZnJhbmsuZGVuaXNA
Y29ycC5vdmguY29tPokCNgQTAQgAIAIbAwIeAQIXgAUCVNnRegQLCQgHBRUKCQgL
BRYCAwEAAAoJECEGJ6q6cJ/h0LgP+wfCw2SCFvD7sFlnmd6oJNP+ddtt+qbxDGXo
UbhrS1N88k6YiFRZQ+Z84ge9RgQXA74xuWlx8g1YBEsqO1rYCGQ4C+Ph+oUO+a3X
k+wmEzINnjCF8CQzZQ3vdXvWmshKzqC2yyeR235WC/BSHsqsr+TRFEmGa68ju8s7
UF8ZQaBzbM0ttUtrc0UqhnS16xV5lH9gBkVbMWIN1pAeJcFRL6MB92Vv5tWjayua
w76vxmwPhu6quUlwxNYNvYBgG5kpBjqMOLHaX1x+SA5F6aI6E3kqxeyurwV6Ty+/
FIns+Awl+IFPey5ctwSOXkizhtqxpMNHAu9resNRjneIjNVTLON1uaxvmPJttMd/
CdTXh+guxDBfH6Vr9nmExy2qbihDJ06Sm874UYtnBZdB7Fi0cNF1DlEZKaZyYaLw
RA/TelI2IaIdkRFLsaFdo144nfceZ2fra2QO83Ow6uShNZzAHU0ZVEKLVt/VJqCL
6hts7vhKuCBcNlpoNOZptRPJf8RMLh4qwtniZadDcM16TpvkyTQUAWH+GvTML0UR
5sLHOtZ7MUaHO/c5UWQWJOmuaWOKgdKLi3iXztGbNNDc9F7wRoObUH7Om/0s5IRy
noO58ofDCmurPDP+10eOQaWtgVz2nFXcFF0qTw4H6L/sXlzbm27HuqEHuYrzpTl/
Njn0chjBiEYEEBEIAAYFAlTZ0RwACgkQkmvFFxzepDnrmQCfdaiJcQsAZaSfEfO1
VxZaY0kEVf0An1xVULYvo5M4sta0tILFu3UthzBGtDdGcmFuayBEZW5pcyAoSmVk
aS9TZWN0b3IgT25lKSA8MGRheWRpZ2VzdEBwdXJlZnRwZC5vcmc+iQI2BBMBCAAg
BQJU2dKRAhsDBAsJCAcFFQoJCAsFFgIDAQACHgECF4AACgkQIQYnqrpwn+FqRxAA
wWm+f6mo9nCoGRD4r4jrSLuJ5ApyIxRQ3L5DL/MeITRMPNDps0OpvKIIGmGv19n5
Ani7ufOcnQLkTVj1179U5BTnahk2fDS0CxlFyslpR9A7tX6qQMtIyBE4cdPhjVue
ZOwI+PfJSleFFmPQ3ESlbKzeNGJqBQiNSbpo9qMhhyYRZy/Fk4kOQzAdXpa63kPX
1KVoTsvz19O2frLim7QY8oTI8Vbij0CB+HfhHuLmolc039/S47hF+5ygERK5Fwjo
mSx+Q2fKx9P35TZqQ9Zw73e3gS9YUErT4LU7ZwdmulftfCaVLmIuX4GUDPasmNbA
WLpKHEwLln0YJO0kIzD+2q2zclzUmGgdgGcEUwLb6vpWLJ41MsmHknZg0zm/yG6/
sasA0jU1wKxeRlHeSxnh3PYb+v36kHXsRViqPlwxe9PGmLK9p9wD0yS/dk2LsJbE
1hnUZfw7l14VdivrL567My/0sG3SbIUb/DxHuVkgHU9LHHlca4z5VmFc7v2+sc0+
6IczFW86FKI8m+q8zLhHcquKgZpumxvwjEoAbjl9123bqZKm1e8pHL3bTQa6bSv9
isNsW3T9eHeEB7frbBlYOZjvMQuYLf82t2tu+E4xbUYZZrmlRYGwBGFUBRprtJ0e
XeUvxFgAnazyNNXxXhO3PMiCxpCp0e7+x64fKVPMfFu5Ag0EVNnQDwEQAMnv/UG9
7vAtIyeG+lPalmhn10NQ07I4Rz+vigZHAxO8t7QYhOYOYLZFj1mO11f8lc5X1oxV
7dKwh+sHMJQ3fkOmQbG6VGRLmRTAPk45GsaRcAnczNzCZWw0s4f92ybc9Th4dNR8
dUk90t+tFItPGnFHGHmjwUYMc7u8BNl9l/SNiJipxuHjUR1hXQE+RXrlgkoW9S8I
bisHytd5IcOXGz337coYkdJLzx1AdpOMGN4n5qymlrhjBIvV2a/R+mweUAD7Il8I
Ynj58lalrp2kLmnoJacL0R9R2ZbSjDBevKpitmy3kqHS59vChw80asBRWr10++Ea
V0LnWDKKbc1U809RP1Ac0l66KjKj3mmiQQKDpb2oHHD0uJsx84kqCOkoWdqF12wR
stygYsAc8CJXnsAKThdDvsQTkMX6WKg4wtSJw0ELRtNCQZzH8iE6eq9MXZijvG6H
j9WyZ2L2eeO0bKn0uEDGvpPMLWcFfOjCxL32x/Jr95sqAt2p0DcBFH5d4jK7tqHQ
YzNwt8ibbbGlwzRFTgq/5igV+n9q9P/h8bWQhUJyqbjyJuwt4l/oTSTKZ5bZ0IAr
KS/+Y/Y9b/BBXRzRP/D1LhaOndH43E6HmEWGS2PhUUPn3V6TQzOq5npaTXKhq/f8
XMYEqvbQ3qjfREa+LLgmFLAwD7rc8h2WYVp7ABEBAAGJAh8EGAEIAAkFAlTZ0A8C
GwwACgkQIQYnqrpwn+GCVhAAscO0pYCRzcgDwDWOrT3g5yi8dt3NmDGL9c6/ohKV
waWSIDlwFtbZNiZ/fr91VCdDfhUSohtn6E7XvKYdVNO4NRLIbSgRc7Y/C4P+9lEh
k+6mlXYlEil/GN6YXBsQvDSz1xw+Csz3Y6kq2m1xiSHFuZrP0PS75x+vIAKbIspa
uu5IyEh/wAW1vY/pnzs7TJtY2r8Qsv/5xt+zUdlGB0ZJq7IZ/1GveltRMJrfhcCT

KPQRWdMv0aEioeBwYAM8sc9UrrePM9jSpT3uCYwuJlld4M94+tqt7tqvkR6dluXF
+4WWeuPXo65jSBl094BEfT5dVbt0TqmG6eTgnPghh1j7PpIghyqUU0v8YPl5DUnZ
UuHzi4CEcQWNUEq+xK9N2/nflaq8R4LPDJjupSWIw5tZv8NWj+EA/zyxggX+q2pr
3qlD+IUnO8cR/RT1LvZ9L5t1fvTqjpgDqXJIremihObLOGEV0+0xWEaN085OVzyU
QTt2EBhzSxHkC0CEd6CgR8l48YGsKJrHCjuOvQ+lgVtAkgYBeVFefhrKa242TmVB
NlZCkS25wUhGhWbLv334p+KTG4d79J+iKYbh8n0C/gBK0YzDX3gLbL+6wes0xYia
WSRBfx9hfPCfFLDGG5sY7yViH8YcOGig6IV9+DWBCSyOZ0d0IXWNvTLF+3d1BFD4
dlG5Ag0EVNnQNwEQANZNoFI4cM9TYFCMOYIiH1UaXoibNE7kZ1qDM/O6y5HTUOSn
m2koCYMTqtVaigAq/tXiUJLBzoHwh17CzDx5L3/IShMHdqwAFCcUZII2NW/XEEH7
knwnqn5tki2CZCzfE+GXtUm7M7fBW2pgPvVt/Ord+DhmEKP0A+fdKHS3x/EUn8Vs
vJoYEkxg9fT14eqYk+oALFxm6vW9UAFO0VZ/JOXzeDTux0+6p6NQjcykKeG5GiXA
dHpRopfeksLQx3sZqfFBEhuiIX7PllAQxHpPqKcPG82aVqT5x9tvZ2RVdk/55hcK
gNhdcbDGWqkNENbOvTmom2a/gDNgb7pf12jJa9t2RRVC8oyYh+zVftLhf2GlwMVv
vwuXO1U2A0/lUQ7K33t6lQ2mEmbudyeFJCso3kIJ598efTw2ZPkeEkZ+adsIBQbd
CSEm0B/S+DS8CDTLTfS5nN5T3rGnO7lzPf983uP9CLbODyt05dqF1Hl+4XicMT3P
Qtz1T+P7X7nPQL9FUwOWUBHqfhYhNsnV17m6M/ODoKsyjdl92njOxvyD6zVaffcx
2zX+SYEaIIiDFhxVFprhwTuruKOfax3nNTLd1JeiraUejSNCnP60VxTsp203Y0H8
quLtvsWF6V5lr57WQxGQxQmS5JQV9wreYzuA339ApUqukfWmhiPDHbQVWAe3ABEB
AAGJBD4EGAEIAAkFAlTZ0DcCGwICKQkQIQYnqrpwn+HBXSAEGQEIAAYFAlTZ0DcA
CgkQYvJbWStvdtq1jg/8Dm6BicjEbcNphWpsjj0uoPB49I0fKFxSM2uUh6PI+wtc
LtikJsNyGvXDm7oGE/uXIki5S++91pZ5oTV931HVzp8e4vip5IRCcWFk6NisRmiZ
nN/xMejLnK3s51pmK5UJhoYymrETGiUKj1uu5BqewRXZ4wWH2kzIusBzIc537shR
Gqk+LgwY7/x4aKY+5Z46VpAGSlO4a6WdWtlRLZzOz0x+tPIrAYo0f72hdHg2enZE
rqkhi90dy/5hCsaJRl+raEZVDSggOtO0hmhTnLSWAX3YPINp1qSqvn5EQk8FhZuh
RaonpXg0wZLc82oIYEZ0KnhJ7HBgV/jF78lI5ZPdk9m22GbASWkIjwNmfzAhGEPu
/NX3iweDPfU4ULbOvejs3ivQTEOrF47u3ps/6SOrBXS7f23ZBw7nwYryezCeQUV8
RCKkk+xUPv5YU0DpGtViDrfxeucXW8W05VOBsCfpa2PTXvj4VjP6UGRUcX3SVTcA
VnvKAmfsDa/4+4AOEvfgQFRzuex8tthFbPW2pLJEQPpVFuxAK0foUHw78HFL7NRV
TFx3jUWgGAM7PA9FI9h1rrU5dXyi8uXwBjaXcEaIts7WE0NGjFzEbub6kJldryhl
5ZCMkmOcBU7SkSmI95bOJwvYdGGiEcO4eh7ci4pOFH0ZNqKfpjyfpTgtFgS5Ldne
pBAA8ubnR6+b7gGaOQk/rROTYHoSq9GXVAqhhmY69lfsXQ9EXoiAzNZnhJLtj1J7
86Z3Bgd9X+MXrrPoJLVGmBTT8yT337KY/+rbk16E5oL1eItnsJ0xgprD1gkWUNaa
pRXLKdA86ogoU8sE/9Wr2CN6dCdPCmjmc0mWvGHY5V6lMf3NPIAQbS4izuU/w+IE
gPnBo45BPkxP2HyvhoOek+pxpsqL8uLQzuIjtwgWvMOocVQrpBNr6kQ99hvr8feY
6kOI5MoGsagW3R65m7DAfz/x1oO3QmWT/kg2dcWqiEbzL3phX1QpQtdJkO5+JTYQ
F0WP5sPzQ7DaIP7Mo2NjhqvnO5NR9/kEzX1yEQck3BI4vKNHSiAQ1/J94uiu9Aze
W6ddPO4Ax7LycK0WOeNVNAT6a3tFJbQrve3ZoDDSNXAa70VKmpdrsrwnX+/4+rly
Z7lj7rnMWCe9jllfZ2Mi+nIYXCrvhVh0t7OHVGwpSq28B/e2AFsQZxXcT4Y+6po7
aJADVdb+LlOAuF6xB3sylE1Im0iADCW9UAWub1oiOr9jv0+mHEYc3kaF0kPU5zKO
I9cg891jcOBV/qRv89ubSHifw9hTZB0dDjXzBjNwNjBHqkYDaLsf1izeYHEG4gEO
sjoMDQMqgw6KyZ++6FgAUGX5I1dBOYLJoonhOH/lNmxjQvc=
=Hkmu
-----END PGP PUBLIC KEY BLOCK-----