Zinc
Minimal Lightweight Crypto API

Presented by Jason A. Donenfeld

September 26, 2018

Who Am I?

- Jason Donenfeld, also known as **zx2c4**.
- Background in exploitation, kernel vulnerabilities, crypto vulnerabilities, and been doing kernel-related development for a long time.
- Have been working on WireGuard – an in-kernel VPN protocol – for the last few years.

WireGuard

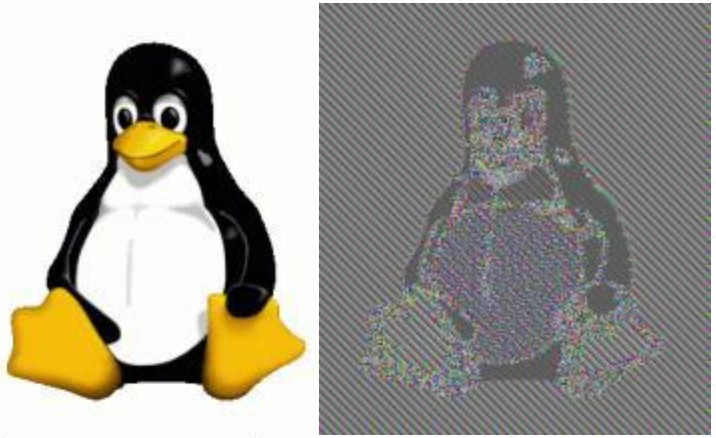
- Less than 4,000 lines of code.
- Easily implemented with basic data structures.
- Design of WireGuard lends itself to coding patterns that are secure in practice.
- Minimal state kept, no dynamic allocations.
- Stealthy and minimal attack surface.

Crypto API Doubts

- Are the WireGuard objectives of simplicity of the codebase and extreme auditability possible with the existing crypto API?

Case study: security/keys/big_key.c

- Stores key in memory, encrypted data on disk. Gives plain-text back to user if user has access to key. (See keyctl(1).)
- Originally the crypto was totally broken.
- Used ECB mode:
- Missing authentication tag – keys could be modified on disk.
- Bad source of randomness.
- Key reuse.
- Improper key zeroing.
- CVEs!



Case study: security/keys/big_key.c

- Seeing that it was broken, I rewrote it, making proper use of the crypto API.

```
static struct crypto_aead *big_key_aead;
static DEFINE_MUTEX(big_key_aead_lock);

// Confusingly passing "CRYPTO_ALG_ASYNC" means "don't be async"!
big_key_aead = crypto_alloc_aead("gcm(aes)", 0, CRYPTO_ALG_ASYNC);
if (IS_ERR(big_key_aead))
        ...
ret = crypto_aead_setauthsize(big_key_aead, ENC_AUTHTAG_SIZE);
if (ret < 0)
        ...
```

Case study: security/keys/big_key.c

```
int ret;
struct scatterlist sgio;
struct aead_request *aead_req;
u8 zero_nonce[crypto_aead_ivsize(big_key_aead)];

aead_req = aead_request_alloc(big_key_aead, GFP_KERNEL); // Have to allocate memory!
if (!aead_req)
»       ...

memset(zero_nonce, 0, sizeof(zero_nonce));
// Using scattergather means data must not be on the stack!
sg_init_one(&sgio, data, datalen + (op == BIG_KEY_ENC ? ENC_AUTHTAG_SIZE : 0));
aead_request_set_crypt(aead_req, &sgio, &sgio, datalen, zero_nonce);
aead_request_set_callback(aead_req, CRYPTO_TFM_REQ_MAY_SLEEP, NULL, NULL);
aead_request_set_ad(aead_req, 0);
```

Case study: security/keys/big_key.c

```
mutex_lock(&big_key_aead_lock);
// The key is a part of the global object, so we have to take a
// mutex before setting it. In other words: we have to allocate
// lots of memory for each different key in use, or take locks.
if (crypto_aead_setkey(big_key_aead, key, ENC_KEY_SIZE))
»       ...

ret = crypto_aead_encrypt(aead_req);
mutex_unlock(&big_key_aead_lock);
aead_request_free(aead_req);
return ret;
```

Case study: security/keys/big_key.c

- Problem: big_key likes to kmalloc around a megabyte worth of material.
- Some systems cannot kmalloc that much.
- Solution: kvalloc? Nope, not with the crypto API.

Case study: security/keys/big_key.c

```
commit d9f4bb1a0f4db493efe6d7c58ffe696a57de7eb3
Author: David Howells <dhowells@redhat.com>
Date:   Thu Feb 22 14:38:34 2018 +0000

    KEYS: Use individual pages in big_key for crypto buffers

    kmalloc() can't always allocate large enough buffers for big_key to use for
    crypto (1MB + some metadata) so we cannot use that to allocate the buffer.
    Further, vmalloc'd pages can't be passed to sg_init_one() and the aead
    crypto accessors cannot be called progressively and must be passed all the
    data in one go (which means we can't pass the data in one block at a time).

    Fix this by allocating the buffer pages individually and passing them
    through a multientry scatterlist to the crypto layer.  This has the bonus
    advantage that we don't have to allocate a contiguous series of pages.

    We then vmap() the page list and pass that through to the VFS read/write
    routines.
```

Case study: security/keys/big_key.c

```
static void *big_key_alloc_buffer(size_t len)
{
        struct big_key_buf *buf;
        unsigned int npg = (len + PAGE_SIZE - 1) >> PAGE_SHIFT;
        unsigned int i, l;

        buf = kzalloc(sizeof(struct big_key_buf) +
                      sizeof(struct page) * npg +
                      sizeof(struct scatterlist) * npg,
                      GFP_KERNEL);
        if (!buf)
                return NULL;

        buf->nr_pages = npg;
        buf->sg = (void *)(buf->pages + npg);
        sg_init_table(buf->sg, npg);

        for (i = 0; i < buf->nr_pages; i++) {
                buf->pages[i] = alloc_page(GFP_KERNEL);
                if (!buf->pages[i])
                        goto nomem;

                l = min_t(size_t, len, PAGE_SIZE);
                sg_set_page(&buf->sg[i], buf->pages[i], l, 0);
                len -= l;
        }

        buf->virt = vmap(buf->pages, buf->nr_pages, VM_MAP, PAGE_KERNEL);
        if (!buf->virt)
                goto nomem;

        return buf;

nomem:
        big_key_free_buffer(buf);
        return NULL;
}
```

Case study: security/keys/big_key.c

- All of this trouble to just encrypt a buffer with the most common authenticated encryption scheme.
- Have to allocate once per encryption.
- Have to allocate once per key.
- Cannot use stack addresses or vmalloc'd addresses.
- Bizarre string parsing to even select our crypto algorithm.
- Super crazy "enterprise" API that is very prone to failure.
- Overwhelmingly hard to use.

Case study: security/keys/big_key.c

- Zinc's fix for this:

```
security/keys/Kconfig    |   4 +-
security/keys/big_key.c | 230 ++++++++------------------------------------------
2 files changed, 28 insertions(+), 206 deletions(-)
```

Case study: security/keys/big_key.c

- Essentially amounts to cleaning out the old cruft, plus:

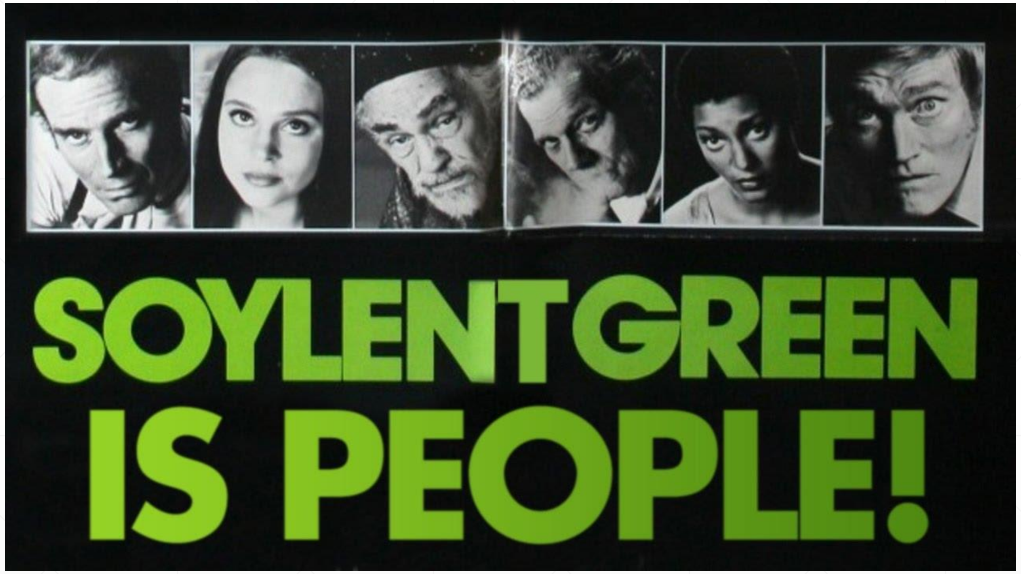
```
buf = kvmalloc(enclen, GFP_KERNEL);
if (!buf)
        return -ENOMEM;

/* generate random key */
enckey = kmalloc(CHACHA20POLY1305_KEY_SIZE, GFP_KERNEL);
if (!enckey) {
        ret = -ENOMEM;
        goto error;
}
ret = get_random_bytes_wait(enckey, CHACHA20POLY1305_KEY_SIZE);
if (unlikely(ret))
        goto err_enckey;

/* encrypt data */
chacha20poly1305_encrypt(buf, prep->data, datalen, NULL, 0,
                         0, enckey);
```

Zinc is Functions!

- Not a super crazy and abstracted API.
- Zinc gives simple functions.
- High-speed and high assurance software-based implementations.
- **Innovation:** C has functions!



Zinc is Functions!

- ChaCha20 stream cipher.
- Poly1305 one-time authenticator.
- ChaCha20Poly1305 AEAD construction.
- BLAKE2s hash function and PRF.
- Curve25519 elliptic curve Diffie-Hellman function .
- We're starting with what WireGuard uses, and expanding out from there.

Real World Example: Hashing

- One shot:

```
blake2s(mac1, message, key, COOKIE_LEN, len, NOISE_SYMMETRIC_KEY_LEN);
```

- Multiple updates:

```
struct blake2s_state blake;

blake2s_init(&blake, NOISE_SYMMETRIC_KEY_LEN);
blake2s_update(&blake, label, COOKIE_KEY_LABEL_LEN);
blake2s_update(&blake, pubkey, NOISE_PUBLIC_KEY_LEN);
blake2s_final(&blake, key, NOISE_SYMMETRIC_KEY_LEN);
```

Zinc is Functions!

- This is not very interesting nor is it innovative.
- These are well-established APIs.
- It is new to finally be able to do this in the kernel.
- No domain-specific string parsing descriptor language:
  - "authenc(hmac(sha256),rfc3686(ctr(aes)))"
- Very straightforward.

Zinc is Functions!

- Dynamic dispatch can be implemented *on top of* Zinc.
  - Existing crypto API can be refactored to use Zinc as its underlying implementation.
- Tons of crypto code has already leaked into lib/, such as various hash functions and chacha20. Developers want functions! Zinc provides them in a non haphazard way.

Implementations

- Current crypto API is a museum of different primitives and implementations.
- Who wrote these?
- Are they any good?
- Have they been verified?

Implementations

- Zinc’s approach is, in order of preference:
  - Formally verified, when available.
  - In widespread use and have received lots of scrutiny.
    - Andy Polyakov’s implementations, which are also the fastest available for nearly every platform.
  - Stemming from the reference implementation.

Implementations

- ChaCha20: C, SSSE3, AVX2, AVX512F, AVX512VL, ARM32, NEON32, ARM64, NEON64, MIPS32
- Poly1305: C, x86_64, AVX, AVX2, AVX512F, ARM32, NEON32, ARM64, NEON64, MIPS32, MIPS64
- BLAKE2s: C, AVX, AVX512VL
- Curve25519: C, NEON32, x86_64-BMI2, x86_64-ADX
- **Super high speed.**

Formal Verification

- HACL* and fiat-crypto
- Machine-generated C that's actually readable.
- Define a model in F* of the algorithm, prove that it's correct, and then lower down to C (or in some cases, verified assembly).
- Much less likely to have crypto vulnerabilities.
- HACL* team is based out of INRIA and is working with us on Zinc.

Stronger Relations with Academia

- People who design crypto primitives and the best and brightest implementing them generally don’t come near the kernel:
  - It’s weird, esoteric, hard to approach.
- Goal is to make this an attractive project for the best minds, to accept contributions from outside our kernel bubble.
- Several academics have already expressed interest in dedicating resources, or have already begun to contribute.

Fuzzing

- All implementations have been heavily fuzzed and continue to be heavily fuzzed.

```c
int LLVMFuzzerTestOneInput(const unsigned char *input, unsigned long len)
{
	unsigned char out1[16], out2[16], out3[16];
	unsigned char key1[32], key2[32], key3[32];
	unsigned char in1[256], in2[256], in3[256];

	if (len < 32 || len > 130)
		return 0;

	memcpy(key1, input, 32);
	memcpy(key2, input, 32);
	memcpy(key3, input, 32);
	memcpy(in1, input + 32, len - 32);
	memcpy(in2, input + 32, len - 32);
	memcpy(in3, input + 32, len - 32);

	poly1305_hacl128(out1, in1, len - 32, key1);
	poly1305_hacl256(out2, in2, len - 32, key2);
	poly1305_donna32(out3, in3, len - 32, key3);

	assert(!memcmp(out1, out3, 16) && !memcmp(out2, out3, 16));

	return 0;
}
```

Assurance

- By choosing implementations that are well-known and broadly used, we benefit from implementation analysis from across the field.
- Andy Polyakov’s CRYPTOGAMS implementations are used in OpenSSL, for example.

Straightforward Organization

- Implementations go into lib/zinc/{name}/
  - lib/zinc/chacha20/chacha20.c
    lib/zinc/chacha20/chacha20-arm.S
    lib/zinc/chacha20/chacha20-x86_64.S
- By grouping these this by primitive, we invite contribution in an approachable and manageable way.
- It also allows us to manage glue code and implementation selection via compiler inlining, which makes things super fast.
  - **No immense retpoline slowdowns due to function pointer soup.**

Compiler Inlining

```
static inline void poly1305_emit(void *ctx, u8 mac[POLY1305_KEY_SIZE],
				 const u32 nonce[4],
				 simd_context_t *simd_context)
{
	if (!poly1305_emit_arch(ctx, mac, nonce, simd_context))
		poly1305_emit_generic(ctx, mac, nonce);
}
```

Branch Prediction is Faster than Function Pointers

```
static inline bool poly1305_emit_arch(void *ctx, u8 mac[POLY1305_MAC_SIZE],
                                      const u32 nonce[4],
                                      simd_context_t *simd_context)
{
#if defined(CONFIG_KERNEL_MODE_NEON)
        if (poly1305_use_neon && simd_use(simd_context)) {
                poly1305_emit_neon(ctx, mac, nonce);
                return true;
        }
        convert_to_base2_64(ctx);
#endif

        poly1305_emit_arm(ctx, mac, nonce);
        return true;
}
```

SIMD Context Optimizations

- Traditional crypto in the kernel follows usage like:

```
kernel_fpu_begin();

while (walk.nbytes >= CHACHA20_BLOCK_SIZE) {
        chacha20_dosimd(state, walk.dst.virt.addr, walk.src.virt.addr,
                        rounddown(walk.nbytes, CHACHA20_BLOCK_SIZE));
        err = skcipher_walk_done(&walk,
                                 walk.nbytes % CHACHA20_BLOCK_SIZE);
}

if (walk.nbytes) {
        chacha20_dosimd(state, walk.dst.virt.addr, walk.src.virt.addr,
                        walk.nbytes);
        err = skcipher_walk_done(&walk, 0);
}

kernel_fpu_end();
```

SIMD Context Optimizations

- What happens when encrypt is called in a loop?

```
for (packet in packets) {
        encrypt(packet);
}
```

- We have to save and restore the FPU registers every time.
- Super slow!

SIMD Context Optimizations

- Solution: simd batching:

```
simd_context_t simd_context;

simd_get(&simd_context);
for (packet in packets) {
    encrypt(packet, &simd_context);
    simd_relax(&simd_context);
}
simd_put(&simd_context);
```

- Familiar get/put paradigm.
- Since simd disables preemption, simd_relax ensures that sometimes we do toggle simd on and off.

SIMD Context Optimizations

- Then, the crypto implementations check simd_use, to activate simd (only the first time):

```
void encrypt(struct packet *packet, simd_context_t *simd_context)
{
    if (packet->len >= LARGE_FOR_SIMD && simd_use(simd_context))
        wild_simd_code(packet);
    else
        boring_scalar_code(packet);
}
```

- Avoids activating simd if it’s not going to be used in the end.

Zinc: Lightweight and Minimal

- Change in direction from present crypto API.
- Faster.
- Lightweight.
- Easier to use.
- Fewer security vulnerabilities.
- Maintained by Jason Donenfeld (WireGuard) and Samuel Neves (BLAKE2, NORX, MEM-AEAD).
- Currently posted alongside WireGuard in v6 form.
- We’re shooting for Linux 5.0.

Jason Donenfeld

- Personal website: www.zx2c4.com
- WireGuard: www.wireguard.com
- Company: www.edgesecurity.com
- Email: Jason@zx2c4.com