THE PARIS256 ATTACK

Or, Squeezing a Key Through a Carry Bit.

Sean Devlin, Filippo Valsorda

Introduction

We present an adaptive key recovery attack exploiting a small carry propagation bug in the Go standard library implementation of the NIST P-256 elliptic curve, reported to the Go project as issue 20040.

Following our attack, the vulnerability was assigned CVE-2017-8932, and caused the release of Go 1.7.6 and 1.8.2.

The affected code

The Go standard library includes an assembly implementation of the NIST P-256 elliptic curve for the x86-64 architecture. One of the fundamental building blocks of such an implementation is constant time arithmetic modulo p (the prime number $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$).

Here's the original `p256SubInternal`, the function computing $a = a - b \bmod p$ in constant time. Some registry aliases were renamed for clarity.

```
TEXT p256SubInternal(SB),NOSPLIT,$0
    XORQ mul0, mul0

    SUBQ b0, a0
    SBBQ b1, a1
    SBBQ b2, a2
    SBBQ b3, a3
    SBBQ $0, mul0

    MOVQ a0, t0
    MOVQ a1, t1
    MOVQ a2, t2
    MOVQ a3, t3

    ADDQ $-1, a0
    ADCQ p256const0<>(SB), a1
    ADCQ $0, a2
    ADCQ p256const1<>(SB), a3

    ADCQ $0, mul0

    CMOVQNE t0, a0
    CMOVQNE t1, a1
    CMOVQNE t2, a2
    CMOVQNE t3, a3

    RET
```

This is Plan9 assembly, where data "flows" from left to right. So, for example, `SUBQ b0, a0` means $a0 = a0 - b0$.

Let's break it down. Notice, first of all, how the numbers we are operating on are 256-bit wide, but our machine registries are only 64-bit. This is why each number is broken into 4 registries. The inputs are a0–a3 and b0–b3, and the result is stored in a0–a3.

- The first series of operations executes the $a = a - b$ subtraction itself.

  `SUBQ` subtracts the least-significant register of the 4, and sets the carry/borrow flag. Each subsequent `SBBQ` instruction (subtraction with

borrow) subtracts two registries accounting for the borrow from the previous instruction. The last `SBBQ` stores the final borrow in `mul0`.

- The `MOVQ` instructions copy a0–a3 to t0–t3.

  If a was bigger than b, this is the final result, so it's copied for later.

- The `ADDQ`/`ADCQ` sequence executes $a = a + p$.

  If instead a was smaller than b, the result would have wrapped around at 2^{256} (4 · 64-bit registries) instead of at p. Adding p corrects the result, as $a - b + p = a - b \bmod p$ without wrapping around.

- Finally, the `ADCQ` and `CMOVQNE` instructions select in constant time between the two results above $a - b$ and $a - b + p$.

  `ADCQ` sets the zero flag based on `mul0`, the borrow result of the $a - b$ operation, and `CMOVQNE` copies t0–t3 back to a0–a3 based on the zero flag. The `CMOV` instructions execute a copy or a no-op based on a condition in constant time, allowing us not to leak with timing whether a was smaller or bigger than b.

The bug

The issue is that `ADCQ $0, mul0` was simply meant to set the zero flag based on `mul0`, but `ADCQ` adds the carry bit from the previous operation as well, which is completely unrelated. This inverts the condition looked at by the `CMOV`. The implementation offsets that by swapping `CMOVQNE` for `CMOVQEQ`, inverting the check as well.

Here's the patch:

```
TEXT p256SubInternal(SB),NOSPLIT,$0
    XORQ mul0, mul0
    SUBQ b0, a0
    SBBQ b1, a1
    SBBQ b2, a2
    SBBQ b3, a3
    SBBQ $0, mul0

    MOVQ a0, t0
    MOVQ a1, t1
    MOVQ a2, t2
    MOVQ a3, t3

    ADDQ $-1, a0
    ADCQ p256const0<>(SB), a1
    ADCQ $0, a2
    ADCQ p256const1<>(SB), a3

-   ADCQ $0, mul0
-
-   CMOVQNE t0, a0
-   CMOVQNE t1, a1
-   CMOVQNE t2, a2
-   CMOVQNE t3, a3
+   ANDQ $1, mul0
+
+   CMOVQEQ t0, a0
+   CMOVQEQ t1, a1
+   CMOVQEQ t2, a2
+   CMOVQEQ t3, a3

    RET
```

Two bugs compensate each other, and all is well. Almost. The carry bit erroneously included is the one generated by adding p to $a - b$. p is so close to 2^{256} that the addition almost always overflows, setting the carry bit (which is what we want it to do, to compensate the underflow in $a - b$). If, however, $a - b$ is less than $2^{256} - p$, the carry bit won't be set, and the `CMUL` will select the wrong result.

Summing up, if a is within $\left[b, b + 2^{256} - p\right)$, `p256SubInternal` returns $a - b + p$ instead of $a - b$. This happens with a probability of $\frac{2^{256} - p}{p} \cong 2^{-32}$ between random a and b values.

It's a carry propagation bug in that it's caused by a carry improperly propagating into the next unrelated operation.

The adaptive bug attack

So, how do we turn a bug in an internal subroutine causing a wrong result once in 2^{32} times into a key recovery attack?

First, we need to understand how the bug affects higher-level operations. `p256SubInternal` is used by `p256PointAddAsm` and `p256PointDoubleAsm`, both used in `ellipic.P256().ScalarMult` (but, interestingly, not in `ellipic.P256().BaseScalarMult`).

`ScalarMult(Qx, Qy *big.Int, k []byte) (x, y *big.Int)` returns the point $k \cdot (Qx, Qy)$ where k is a number in big-endian form. It implements a variant (see below) of a double-and-add algorithm, meaning doublings ($\cdot 2$) and additions ($+Q$) are performed starting from the zero point in an order dictated by the bit patterns of k: a double for each 0 bit, an add and a double for each 1.

The success of the attack relies on the ability to generate points that break for specific double-and-add sequences. For example, a point Q that triggers the bug when $Z + Q \cdot 2 \cdot 2 + Q \cdot 2 \cdot 2$ ($k = 1010...$) is calculated, but not when $Z + Q \cdot 2 \cdot 2 + Q \cdot 2 + Q \cdot 2$ ($k = 1011...$) is.

Given such a bug, we can construct an oracle from protocols that let the attacker submit arbitrary points Q, and observe the result of `ScalarMult(Qx, Qy, k)` with a secret and fixed k. For example, any implementation of ephemeral-static Diffie-Hellman (ES-ECDH) lets the attacker submit Q as their ephemeral share, and performs scalar multiplication by k, the static key. As long as the attacker knows the dlog of Q, they can complete the exchange and test if the protocol completes successfully; if not, the bug has been triggered.

A bit of the key is then leaked by submitting two points that break for two alternative double-and-add sequences, each corresponding to a different bit at a certain position, and checking which one makes the oracle fail.

The rest of the attack progresses adaptively. Once the bit at position n is discovered, a pair of points breaking for each of the values of the bit at position $n + 1$ is generated and submitted. (As an optimization, one point is sufficient for each round, by exclusion.)

This amounts to a full adaptive key recovery attack against ES-ECDH and similar protocol, where the implementation suffers from the bug above.

Implementation details

The Go x86-64 assembly implementation however is not a plain double-and-add algorithm, but uses a windowing method and Booth encoding. This was based on previous work by Gueron and Krasnov in OpenSSL.

The scalar k is split into 5-bit wide windows, Booth encoded. We call each of the resulting sign and value tuples "limbs". For each limb a value between $|1Q \text{ and } 16Q|$ is selected from a precomputation table and added to or subtracted from the running product. If the limb value is zero the operation is skipped in constant time. The product is then doubled 5 times.

Our attack had then to target one limb at a time instead of one bit at a time, using points that trigger the bug either during the addition or subtraction of the value from the precomputation table, or during the following doubles, as long as only one specific table entry caused the bug to be reached.

There are 33 possible limb values, -16 to $+16$, so for each 5 bits of the key we'll need on average 16 candidate points. (Some limb values restrict the possible values of the following limb, making the search more efficient.)

Fuzzing for points

To find points efficiently, we repurposed the optimized implementation as a high-speed fuzzer. `p256SubInternal` was instrumented to set a flag when the conditions for the bug are met. Given a point and the depth of the target limb, the fuzzer:

- computes the product of the known limbs normally, checking that the bug is not triggered prematurely
- performs the addition and doubles operations for each possible value of the target limb
- if only one limb value triggers the bug, flags the point as a candidate

Points are then post-processed by comparing the outputs of the vulnerable and patched codebases for each of the target limb values, to remove the frequent false positives.

Points to be tested were generated efficiently by starting from a random point of known dlog (required to verify if ECDH completed successfully), and adding G at each iteration. Constant time operations were replaced by slightly faster branches as an optimization.

This allowed us to generate targeted candidate points at a sufficiently high speed to perform the attack in practice. See the conclusion for some benchmarks.

Detecting the first limb

Obtaining the value of the first limb requires extra care, as the first limb causes no addition operations—just a selection from the table—and we need to take into account the operations that produce the precomputation table itself.

If for example a point triggers the bug at the 5th double following the selection of the value 3, it will also trigger the bug when 6 and 12 are selected, because they are nothing else than 3 followed by one or two doubles.

What we need is a set of 3 points, one that fails doubling 3 5 times, one that fails at the 6th double, and one that fails at the 7th double. 3 will only trigger with one of them, 6 with two, and 12 with all three, allowing us to distinguish between them.

The same can be done for all first limb values that would cause ambiguity. The attack is complicated by the fact that the second limb might have a zero value, in which case the entry from the table would be doubled 10 times. A different set of points can be built to detect first limb values when the second limb is zero, and it becomes less and less likely that more consecutive limbs will be zero.

What this attack can't detect is the presence of leading zero-value limbs, as those are silently skipped by the constant time code. This case just takes some care to detect while the attack is terminating.

Thankfully, all the points for this phase of the attack can be precomputed once, as there are no previous limbs to adapt to.

Conclusion

A target: go-jose

The attack wouldn't be complete without a practical target. Thankfully, the JSON Object Signing and Encryption (JOSE) specification allows developers to choose from and combine a dizzying array of cryptographic primitives including ES-ECDH.

We tested our attack against a simple server accepting encrypted JWT tokens, and decrypting them with the library github.com/square/go-jose (through no fault of its own), compiled on Go 1.8.1.

Tokens that triggered the bug would cause a different error, acting as an oracle for our adaptive attack, leading to full key recovery.

Benchmarks

We estimate that we find a useful candidate point every 2^{26} tested points. Each limb requires on average 16 candidate points, which on 3.1GHz Haswell platform take about 85 CPU hours.

At 52 limbs per key, the attack requires less than 5000 CPU hours, which can be bought from cloud providers for less than $50 in preemptible ("spot") instances.

The attack requires on average a bit more than 800 oracle invocations.

Prior work

Even though we were not aware of it at the time we developed ours, we'd like to acknowledge the extremely similar attack published by Brumley, et al. in "Practical realisation and elimination of an ECC-related software bug attack" (2011). While the implementation details are different as they were targeting a different bug in OpenSSL from 2007, similarly considered not exploitable, the approach matches our adaptive attack.