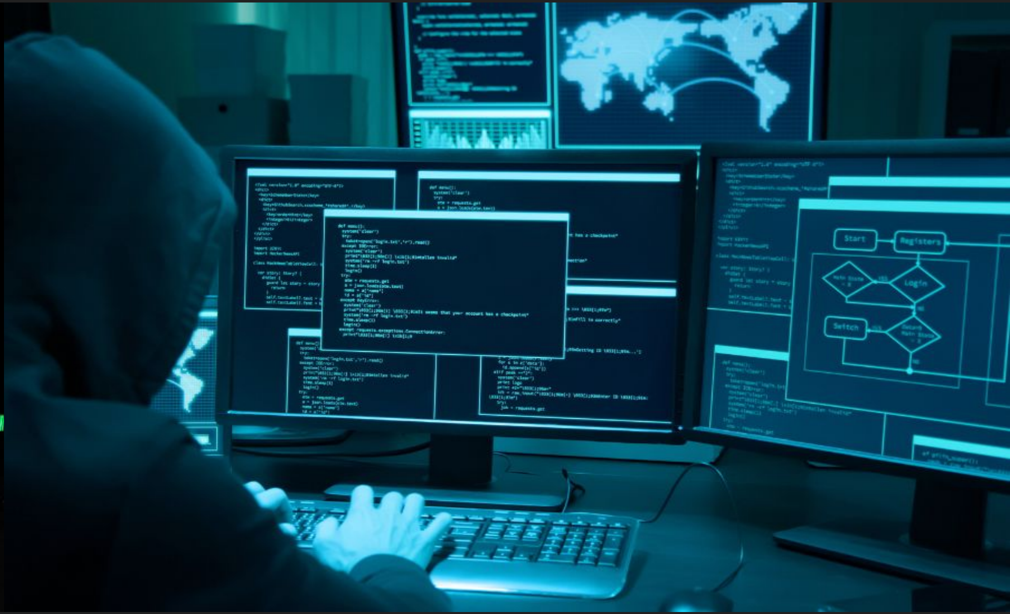
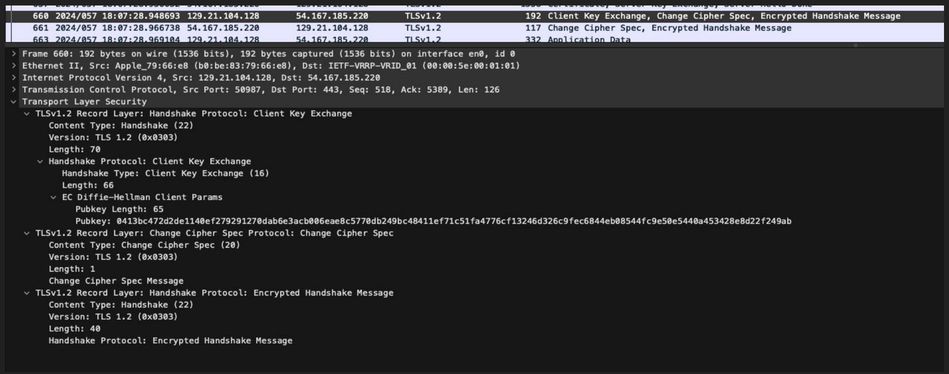
# CVE DEMO

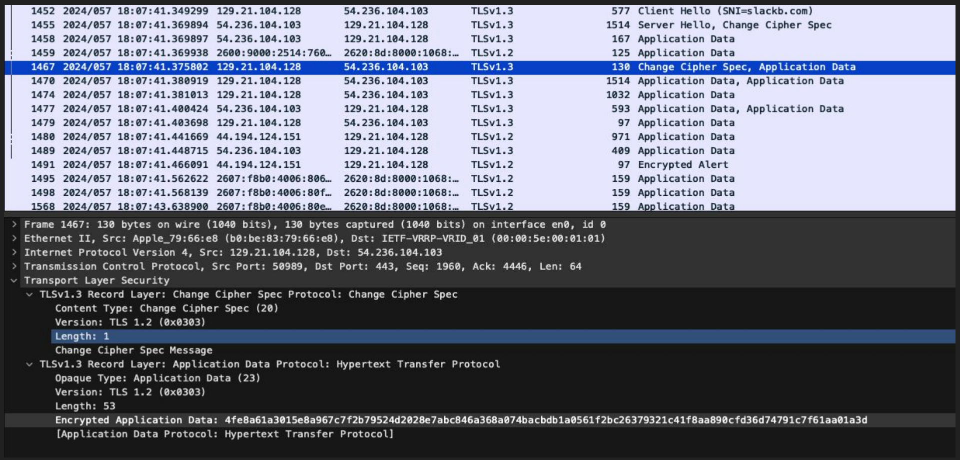



Jonathan Bateman, Hassan Alshehri, Harrison Tarsia

# Why is CVE-2011-4354 important?

- OpenSSL version 0.9.8g
- Implementation vs Efficiency → The Programmers Dilemma
- TLS v1.2 → Why it enables this attack?
  - "Change Cipher Spec" comparison Left (1.2) vs Right (1.3)



# ECDHE Overview

- $k^i_C$ & $k^i_S$ → private keys {1 → (n -1)}
- G → generator
  - (initial elliptic crv point)
- $Q^i_C$ & $Q^i_S$ → public keys (over net)
  - Q = [k]G (private key * generator)
- bug?

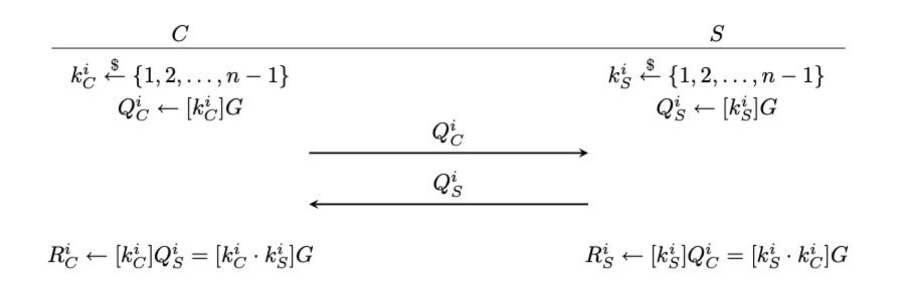
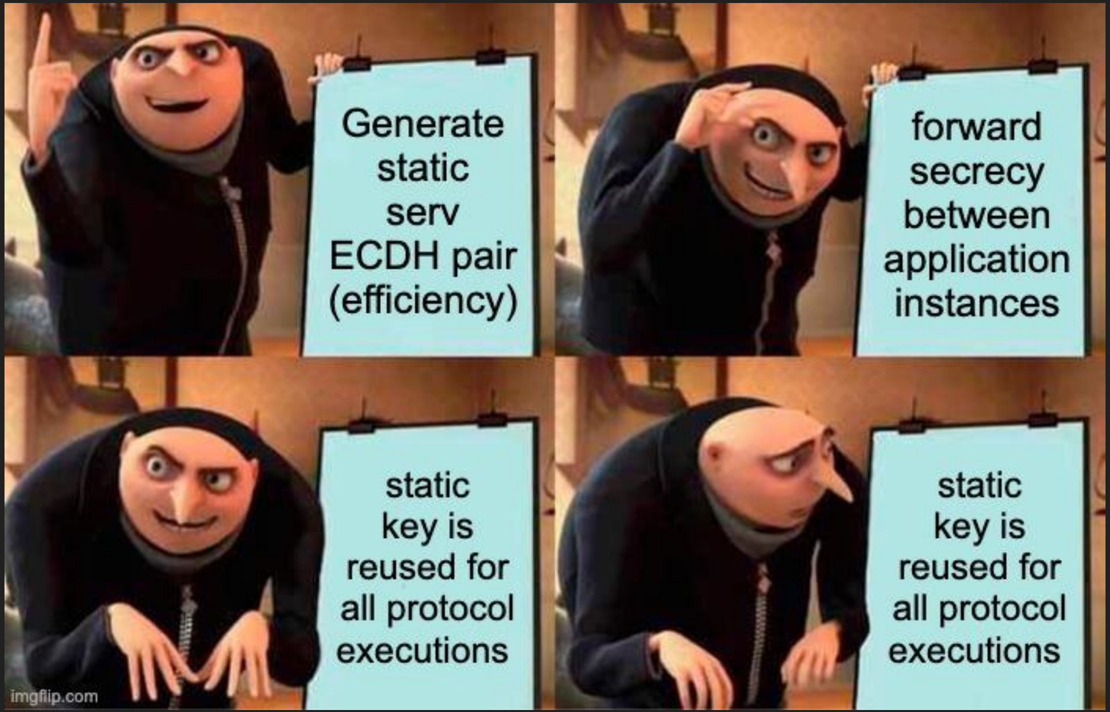



**Fig. 4.** A description of ECDH key exchange.

- $R^i_C$ & $R^i_S$ = $[k^i_S * k^i_C]G$ → multiply personal private key by opposing public key
  - Shared secret → both client and server have it w/o each other's private key
- Attacker: $Q^i_S Q^i_C = k^i_S * k^i_C * G * G \neq [k^i_S * k^i_C]G$ → Very different

# Attack Angle → blame devs who prioritize optimization




$k^i_s$

Stays the same!

# Faulty Algorithm

- The algorithm is used to compute the result of a modular reduction making sure that the quotient is exact and getting rid of the possibility of overflow.
- The problem is the logic of the algorithm is flawed.
- The algorithm is built to handle modular reduction of S by p where c is the exact quotient of the division.
- It is assumed that S = t+c*2^256 and the algorithm tries to compute the quotient
  q = S/p based on the value of c.
- This incorrect reasoning produces erroneous results.

```
nist_cp_bn_0(buf, a_d + BN_NIST_256_TOP, top - BN_NIST_256_TOP, BN_NIST_256_TOP);

/*S1*/
nist_set_256(t_d, buf, 15, 14, 13, 12, 11, 0, 0, 0);
/*S2*/
nist_set_256(t_d2,buf, 0, 15, 14, 13, 12, 0, 0, 0);
if (bn_add_words(t_d, t_d, t_d2, BN_NIST_256_TOP))
  carry = 2;
/* left shift */
  {
  register BN_ULONG *ap,t,c;
  ap = t_d;
  c=0;
  for (i = BN_NIST_256_TOP; i != 0; --i)
    {
    t= *ap;
    *(ap++)=((t<<1)|c)&BN_MASK2;
    c=(t & BN_TBIT)?1:0;
    }
  if (c)
    ++carry;
  }

if (bn_add_words(r_d, r_d, t_d, BN_NIST_256_TOP))
  ++carry;
/*S3*/
nist_set_256(t_d, buf, 15, 14, 0, 0, 0, 10, 9, 8);
if (bn_add_words(r_d, r_d, t_d, BN_NIST_256_TOP))
  ++carry;
/*S4*/
nist_set_256(t_d, buf, 8, 13, 15, 14, 13, 11, 10, 9);
if (bn_add_words(r_d, r_d, t_d, BN_NIST_256_TOP))
  ++carry;
/*D1*/
nist_set_256(t_d, buf, 10, 8, 0, 0, 0, 13, 12, 11);
if (bn_sub_words(r_d, r_d, t_d, BN_NIST_256_TOP))
  --carry;
/*D2*/
nist_set_256(t_d, buf, 11, 9, 0, 0, 15, 14, 13, 12);
if (bn_sub_words(r_d, r_d, t_d, BN_NIST_256_TOP))
  --carry;
/*D3*/
nist_set_256(t_d, buf, 12, 0, 10, 9, 8, 15, 14, 13);
if (bn_sub_words(r_d, r_d, t_d, BN_NIST_256_TOP))
  --carry;
/*D4*/
nist_set_256(t_d, buf, 13, 0, 11, 10, 9, 0, 15, 14);
if (bn_sub_words(r_d, r_d, t_d, BN_NIST_256_TOP))
  --carry;

if (carry)
  {
  if (carry > 0)
    bn_sub_words(r_d, r_d, _256_data + BN_NIST_256_TOP
      --carry, BN_NIST_256_TOP);
  else
    {
    carry = -carry;
    bn_add_words(r_d, r_d, _256_data + BN_NIST_256_TOP *
      --carry, BN_NIST_256_TOP);
    }
  }

r->top = BN_NIST_256_TOP;
bn_correct_top(r);
if (BN_ucmp(r, field) >= 0)
  {
  bn_sub_words(r_d, r_d, _nist_p_256, BN_NIST_256_TOP);
  bn_correct_top(r);
  }
bn_check_top(r);
```

The intention is to eliminate any possibility of overflow; the assumption is that $c$ is the exact quotient of division of $S$ by $p$.

The reasoning behind the faulty algorithm is that if one writes $S = t + c \cdot 2^{256}$, then the exact quotient $q = S \div p$ is given by

1. if $c \geq 0$, then $q = c$ or $q = c + 1$,
2. if $c < 0$, then $q = c$ or $q = c - 1$

since $c$ is small. Indeed, write $\Delta = 2^{256} - p$, then after subtracting $c \cdot p$ we obtain

$$S - c \cdot p = t + c \cdot 2^{256} - c \cdot p = t + c \cdot \Delta.$$

Since $-4 \leq c \leq 6$ and $\Delta < 2^{224}$, this shows the result is bounded by $-p < t + c \cdot \Delta < 2p$. The faulty algorithm therefore computes an incorrect result in the following cases:

- If $c \geq 0$, the algorithm fails when $t + c \cdot \Delta \geq 2^{256}$ since it computes $r'$ only modulo $2^{256}$ and not as a full integer (for which the resulting algorithm would have been correct). Note that in this case the correct result would be $r' + \Delta$ and that modulo $p$, the correct result thus is $r' + 2^{256} \pmod{p}$.
- If $c < 0$, the algorithm fails when $t + c \cdot \Delta < 0$. The correct result then depends on whether $(t + c \cdot \Delta) \bmod 2^{256} \geq p$ or not: in the former case, the correct result is $r' - \Delta$, whereas in the latter case, the correct result is given by $r' + 2^{256} - 2\Delta$. Note that although there are two different subcases for $c < 0$, the errors $-\Delta$ and $2^{256} - 2\Delta$ are congruent modulo $p$, i.e. modulo $p$, the correct result is given by $r' - 2^{256} \pmod{p}$.

**Fig. 3.** A program fragment demonstrating the faulty OpenSSL modular reduction algorithm for P-256.

# The Fix

- Fix in version 0.9.8r
- The issues in the algorithm are associated with the modular addition and subtraction operations.
- The algorithm involves multiple modular additions (bn_add_words) and modular subtractions (bn_sub_words).
- The fixes to the addition and subtraction are shown here on the right.
- Utilizes bitwise operations for efficient function selection based on 'carry'.

```c
u.f = bn_sub_words;
if (carry > 0)
    carry = (int)bn_sub_words(r_d,r_d,_nist_p_256[carry-1],BN_NIST_256_TOP);
else if (carry < 0)
    {
    carry = (int)bn_add_words(r_d,r_d,_nist_p_256[-carry-1],BN_NIST_256_TOP);
    mask = 0-(size_t)carry;
    u.p = ((size_t)bn_sub_words&mask) | ((size_t)bn_add_words&~mask);
    }
else
    carry = 1;

mask  = 0-(size_t)(*u.f)(c_d,r_d,_nist_p_256[0],BN_NIST_256_TOP);
mask &= 0-(size_t)carry;
res   = (BN_ULONG *)(((size_t)c_d&~mask) | ((size_t)r_d&mask));
nist_cp_bn(r_d, res, BN_NIST_256_TOP);
r->top = BN_NIST_256_TOP;
bn_correct_top(r);

return 1;
}
```

# The Fix

- The SSL_OP_SINGLE_ECDH_USE option is used to create a new key every single time when using temporary/ephemeral ECDH parameters.
- The paper states that this option must be turned on otherwise the server will “opt out” and use a static key pair for each “application instance”.
- With the option turned on, a new key pair will be generated per “handshake instance” helping to prevent the attack.

When activated, the optimisation means a single ECDH key pair is generated during initialisation of the OpenSSL context within an application. This key pair is reused for all protocol executions thereafter; with ephemeral-static ECDH, OpenSSL has the server use a static key (i.e., a fixed $k_i^S$ and hence $Q_i^S$ for all $i$) for each OpenSSL context. Put another way, the key pair is ephemeral for each application instance and not (necessarily) per handshake instance. While this preserves forward secrecy between application instances, it violates forward secrecy within a single application instance when performing more than a single protocol execution. Interestingly, the default behaviour is the latter: to “opt out” and disable the optimisation, the application must explicitly use the `SSL_OP_SINGLE_ECDH_USE` option during initialisation of the context.

# Other Changes

| [Takanori Yanagisawa has shown how to correctly use pre-computed values. · openssl/openssl@830b887 · GitHub]() | [Fix remaining BN_nist_mod_*. · openssl/openssl@299ab42 · GitHub]() |
| --- | --- |
| • Table structure<br>• Modification in the Reduction Process (‘if (a>>256)’)<br>• Improvements in carry handling<br>• Boundary checks<br><br>Changes are aimed at handling edge cases and improving efficiency | • Function parameter usage<br><br>Modified code: ‘BN_nist_mod_256(BN_ULONG r[4], const BN_ULONG a[6][4], const BN_ULONG p[6][4], BN_CTX *ctx)’<br><br>The BN_CTX parameter indicates change in how the context is handled during modular reduction |