NCC Group Whitepaper

How to Backdoor Diffie-Hellman

June 17, 2016 – Version 1.0

Prepared by
David Wong – Security Consultant

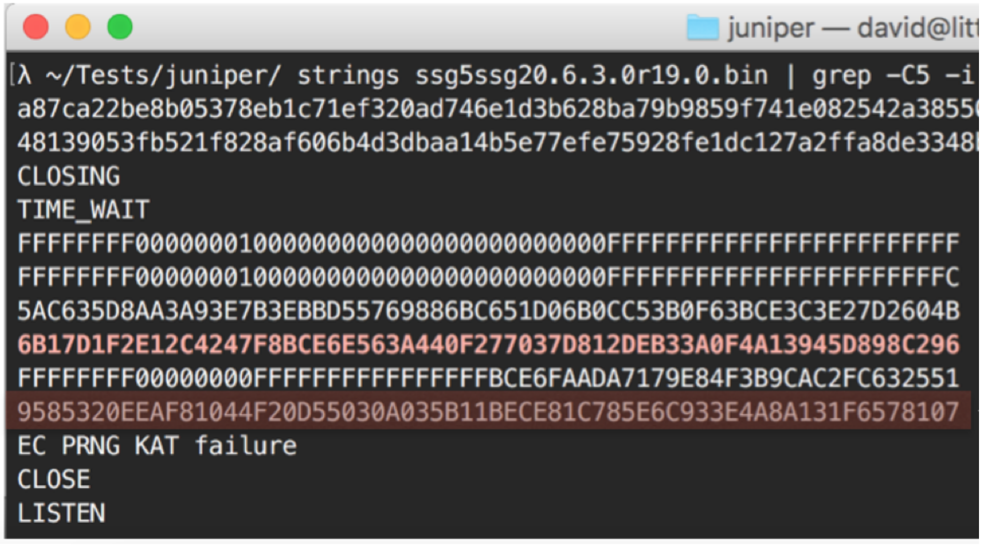
Abstract
Lately, several backdoors in cryptographic constructions, protocols and implementations have been surfacing in the wild: Dual EC in RSA's B-Safe product, a modified Dual EC in Juniper Networks's operating system ScreenOS and a non-prime modulus in the open-source tool socat. Many papers have already discussed the fragility of cryptographic constructions not using nothing-up-my-sleeve numbers, as well as how such numbers can be safely picked. However, the question of how to introduce a backdoor in an already secure, safe and easy to audit implementation has so far received little attention in *public* research.

We present two ways of building a Nobody-But-Us (NOBUS) Diffie-Hellman backdoor: a composite modulus with a hidden subgroup (CMHS) and a composite modulus with a smooth order (CMSO). We then explain how we were able to subtly implement and exploit it in a local copy of an open source library using the TLS protocol.

Table of Contents

1 Introduction

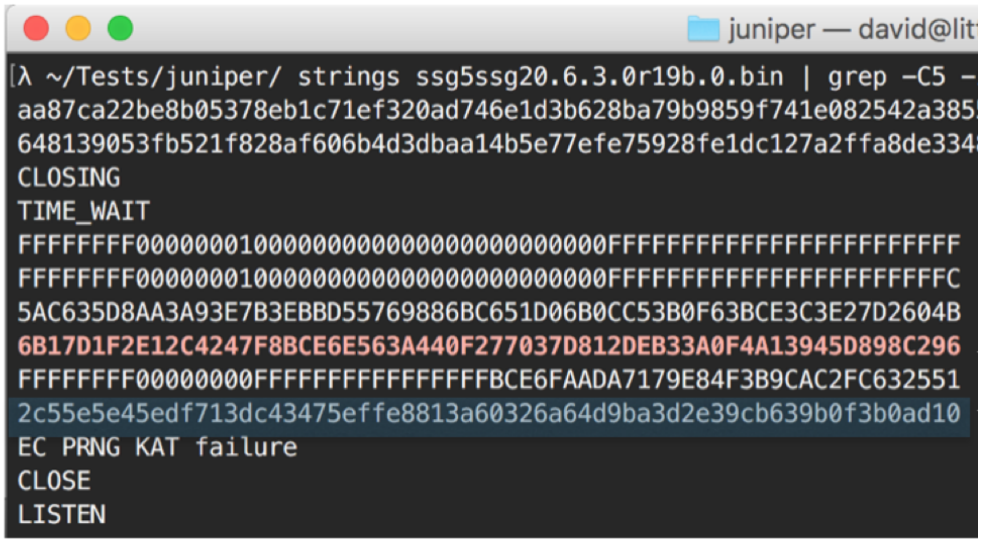
Around Christmas 2015 *Juniper Networks*, a networking hardware company, released an out-of-cycle security bulletin.[1] Two vulnerabilities were disclosed without much details to help us grasp the seriousness of the situation. Fortunately, at this period of the year many researchers were home with nothing else to do but to solve this puzzle. By quickly comparing both the patched and vulnerable binaries, the two issues were pinpointed. While one of the vulnerabilities was a simple "master"-password implemented at a crucial step of the product's authentication, the other discovery was a bit more subtle: a unique value used in the program was modified. More accurately, a number in the source code was replaced. The introduction of the vulnerability was so simple, and due to the fact that the number was stored as a string of hexadecimal digits, the trivial use of the UNIX command line tool *strings* was enough to discover it.



juniper — david@litt

```
[λ ~/Tests/juniper/ strings ssg5ssg20.6.3.0r19.0.bin | grep -C5 -i
a87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a3855
48139053fb521f828af606b4d3dbaa14b5e77efe75928fe1dc127a2ffa8de3348
CLOSING
TIME_WAIT
FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B
6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296
FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
9585320EEAF81044F20D55030A035B11BECE81C785E6C933E4A8A131F6578107
EC PRNG KAT failure
CLOSE
LISTEN
```

Figure 1: The strings of the patched binary



juniper — david@litt

```
[λ ~/Tests/juniper/ strings ssg5ssg20.6.3.0r19b.0.bin | grep -C5 -
aa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a385
648139053fb521f828af606b4d3dbaa14b5e77efe75928fe1dc127a2ffa8de334
CLOSING
TIME_WAIT
FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B
6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296
FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
2c55e5e45edf713dc43475effe8813a60326a64d9ba3d2e39cb639b0f3b0ad10
EC PRNG KAT failure
CLOSE
LISTEN
```

Figure 2: The strings of the vulnerable binary

The special value ended up being a constant used in the system's pseudo-random number generator (PRNG):

[1] https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713

Dual EC, an odd algorithm believed to have been backdoored by the NSA[BLN15]. The PRNG's core has the ability to provide a Nobody-But-Us (NOBUS) trapdoor—a secret passage that can only be accessed by the people holding the secret key. In our case, the elliptic curve discrete logarithm k in the Dual EC equation $Q = [k]P$ (where P and Q are the two elliptic curve points used in the foundation of Dual EC).

Solely the NSA is thought to be in possession of that k value, making them the only ones able to climb back to the PRNG's internal state from random outputs, and then able to predict the PRNG's future states and outputs. The backdoor in Dual EC was pointed out by Shumow and Ferguson[SF07] at Crypto 2007, which might have been the reason why Juniper Networks generated their own point Q in their implementation of Dual EC. Shortly after that revision, a mysterious update would change that Q point one more time, magically allowing another organization, or person, to access that backdoor in place of the NSA or Juniper Networks.

Although the quest to find Juniper Networks's backdoor and the numerous open questions that arose from that work is a fascinating read by itself[CCG+16], it is only the introduction of the work you are currently reading. Here we aim to show how secure and strong cryptographic constructions are a single and subtle change away from being your own secretive peep show.

On February 1st, 2016, only a few months after Juniper Networks's debacle, *socat* published a security advisory of its own[2]:

> In the OpenSSL address implementation the hard coded 1024-bit DH p parameter was not prime. The effective cryptographic strength of a key exchange using these parameters was weaker than the one one could get by using a prime p. Moreover, since there is no indication of how these parameters were chosen, the existence of a trapdoor that makes possible for an eavesdropper to recover the shared secret from a key exchange that uses them cannot be ruled out.

In the same vein as Juniper Networks's problem, a single number was at issue. This time it was the public modulus, an integer used to generate the ephemeral Diffie-Hellman keys of both parties during socat's TLS handshakes. This algorithm had been, contrary to Dual EC, considered secure from the start. But as it turned out, badly understood as well—as the *Logjam*[ABD+15] paper had demonstrated earlier in the previous year, most servers would use Diffie-Hellman key exchanges to perform *ephemeral handshakes*, and the same servers would generate their ephemeral keys from hardcoded defaults (often the same ones) provided by various TLS libraries. The paper raised a wave of discussion around how developers should use Diffie-Hellman, at the same time scaring people away from 1024-bit DH: "We estimate that even in the 1024-bit case, the computations are plausible given nation-state resources".

Securely integrating DH in a protocol is unfortunately not well understood. Defensive approaches are discussed in several RFCs[Res13, Zuc13], but few papers so far have taken the point of view of the attacker. The combination of the current trend of increasing the bitsize of DH parameters with the now old trend of using open source libraries' defaults to generate ephemeral Diffie-Hellman keys would give opportunist attackers a valid excuse to submit their bigger (more secure) and backdoored parameters into open-source or closed-source libraries. This work is about generating such backdoors and implementing them in TLS, showing how easy and subtle the process is. The working code along with explanations on how to reproduce our setup is available on Github.[3]

In Section 2 we will first briefly talk about the several attacks possible on Diffie-Hellman, from small subgroup attacks to Pohlig Hellman's algorithm. In Section 3 we will introduce our first attempt at a DH backdoor. We

[2] http://www.openwall.com/lists/oss-security/2016/02/01/4
[3] https://github.com/mimoo/Diffie-Hellman_Backdoor

will present our first contribution in section 4 by using the ideas of the previous section with a composite modulus to make the backdoor a NOBUS one. In section 5 we will see another method using a composite modulus that allows us to choose a specific generator, allowing us to only modify the modulus value when implementing our backdoor. In section 6 we will explain how we implemented the backdoor in TLS and how we exploited it. We will then see in section 7 how to detect such backdoors and how to prevent them. Eventually we will wrap it all up in section 8.

2 Attacks on Diffie-Hellman and the Discrete Logarithm

To attack a Diffie-Hellman key exchange, one could extract the secret key **a** from one of the peer's public key $y_a = g^a \pmod{p}$. One could then compute the shared key $g^{ab} \pmod{p}$ using the other peer's public key $y_b = g^b \pmod{p}$.

The naive way to go about this is to compute each power of g (while tracking the exponent) until the public key is found. This is called *trial multiplication* and would need on average $\frac{q}{2}$ operations to find a solution (with q the order of the base). More efficiently, algorithms that compute discrete logarithm in expected $\sqrt{q}$ steps like Shank s *baby-step giant-step* (deterministic), *Pollard rho* or *Pollard Kangaroo* (both probabilistic) can be used. Because of the memory required for *baby-step giant-step*, Pollard's algorithms are often preferred. While both are parallelizable, *Pollard Kangaroo* is used when the order is unknown or known to be in a small interval. For larger orders the Index Calculus or other Number Field Sieve (NFS) algorithms are the most efficient. But so far, computing a discrete logarithm in polynomial time on a classical computer is still an open problem.

2.1 Pollard Rho

The algorithm that interests us here is *Pollard Rho*: it is fast in relatively small orders, it is parallelizable and it takes very little amount of memory to run. The idea comes from the birthday paradox and the following equation (where x is the secret key we are looking for; and a, a', b and b' are known):

$$\begin{aligned} g^{xa+b} &= g^{xa'+b'} \pmod{p} \\ \implies x &= (a-a')^{-1}(b'-b) \pmod{p-1} \end{aligned}$$

The birthday paradox tells us that by looking for a random collision we can quickly find one in $\mathcal{O}(\sqrt{p})$. A random function is used to efficiently step through various g^{xa+b} until two values repeat themselves, it is then straightforward to calculate x. Cycle-finding algorithms are used to avoid storing every iteration of the algorithm (two different iterations of g^{xa+b} are started and end up in a loop past a certain step) and the technique of distinguished points is used to parallelize the algorithm. (Machines only save and share particular iterations, for example iterations starting with a chosen number of zeros.)

2.2 Pohlig-Hellman

In 1978, Pohlig and Hellman discovered a shortcut to the discrete logarithm problem[PH78]: if you know the complete factorization of the order of the group, and all of the factors are relatively small, then the discrete logarithm can be quickly computed.

The idea is to find the value of the secret key x modulo the divisors of the group's order by reducing the public key $y = g^x \pmod{p}$ in subgroups of order dividing the group order. The secret key can then be reassembled in the group order using the Chinese Remainder Theorem (CRT), which is described below. The full Pohlig-Hellman algorithm is summarized, with φ being Euler's totient function, as:

1. Determine the prime factorization of the order of the group

$$\varphi(p) = \prod p_i^{k_i}$$

2. Determine the value of x modulo $p_i^{k_i}$ for each i
3. Recompute $x \pmod{\varphi(p)}$ with the CRT

The central idea of Pohlig and Hellman's algorithm is in how they determine the value of the secret key x modulo each factor $p_i^{k_i}$ of the order. One way of doing it is to try to reduce the public key to the subgroup we're looking at by computing:

$$y^{\varphi(p)/p_i^{k_i}} \pmod p$$

Computing the discrete logarithm of that value, we get $x \pmod{p_i^{k_i}}$. This works because of the following observation (note that x can be written $x_1 + p_i^{k_i} x_2$ for some x_1 and x_2):

$$\begin{aligned} y^{\varphi(p)/p_i^{k_i}} &= (g^x)^{\varphi(p)/p_i^{k_i}} \pmod p \\ &= g^{(x_1+p_i^{k_i}x_2)\varphi(p)/p_i^{k_i}} \pmod p \\ &= g^{x_1\varphi(p)/p_i^{k_i}} g^{x_2\varphi(p)} \pmod p \\ &= g^{x_1\varphi(p)/p_i^{k_i}} \pmod p \\ &= (g^{\varphi(p)/p_i^{k_i}})^{x_1} \pmod p \end{aligned}$$

The value we obtain is a generator of the subgroup of order $p_i^{k_i}$ raised to the power x_1. By computing the discrete logarithm of this value we will obtain x_1, which is the value of x modulo $p_i^{k_i}$. Generally we will use the *Pollard Rho* algorithm to compute that discrete logarithm.

The Chinese Remainder Theorem, sometimes used for good[SF] will be of use here for evil. The following theorem states why it is possible for us to find a solution to our problem once we find a solution modulo each power prime factor of the order.

Theorem 1. *Suppose* $m = \prod^k m_i$ *with* $m_1, \cdots, m_k$ *pairwise co-prime.*
For any $(a_1, \cdots, a_k)$ *there exists an* x *such that:*

$$\begin{cases} x = a_1 \pmod{m_1} \\ \vdots \\ x = a_k \pmod{m_k} \end{cases}$$

There is a simple way to recover $x \pmod m$, which is stated in the following theorem:

Theorem 2. *Moreover there exists a unique solution for* $x \pmod m$*:*

$$x = \sum^k a_i * (\prod_{j \neq i} m_j \overline{m}_j) \pmod m$$

with $\overline{m}_j = m_j^{-1} \pmod{m_i}$

At first, it might be kind of hard to grasp where that formula is coming from. But let's see where it does by starting with only two equations. Keep in mind that we want to find the value of x modulo $m = m_1 m_2$

$$\left.\begin{aligned} x = a_1 \pmod{m_1} \\ x = a_2 \pmod{m_2} \end{aligned}\right\} \implies x = \,? \pmod m$$

How can we start building the value of x?

$$\text{If } x = a_1 m_2 \pmod m,$$

$$\text{then } \begin{cases} x = \mathbf{a_1} m_2 \pmod{m_1} \\ x = \mathbf{0} \pmod{m_2} \end{cases}$$

Not quite what we want, but we are getting there. Let's add to it:

$$\begin{aligned}
&\text{If } x = a_1 m_2 \overline{m}_2 \pmod{m} \\
&\overline{m}_2 \text{ the integer congruent to } m_2^{-1} \pmod{m_1} \\
&\text{then } \begin{cases} x = a_1 m_2 \overline{m}_2 = \boldsymbol{a_1} \pmod{m_1} \\ x = 0 \pmod{m_2} \end{cases}
\end{aligned}$$

That's almost what we want! Half of what we want actually. We just need to do the same thing for the other side of the equation, and we have:

$$\begin{gathered}
= a_1 m_2 \overline{m}_2 \pmod{m_1} \\
= a_1 \pmod{m_1} \\
\uparrow \\
\boxed{\begin{array}{l} x = a_1 m_2 \overline{m}_2 + \\ a_2 m_1 \overline{m}_1 \pmod{m} \end{array}} \\
\downarrow \\
= a_2 m_1 \overline{m}_1 \pmod{m_2} \\
= a_2 \pmod{m_2}
\end{gathered}$$

with $\overline{m}_2$ the integer congruent to $m_2^{-1} \pmod{m_1}$ and $\overline{m}_1$ the integer congruent to $m_1^{-1} \pmod{m_2}$.

Everything works as we wanted! Now you should understand better how we came up with that general formula. There have been improvements to it with the Garner's algorithm[4] but this method is so fast anyway that it is not the bottleneck of the whole attack.

2.3 Small Subgroup Attacks

The attack we just visited is a passive attack: the knowledge of one Diffie-Hellman exchange between two parties is enough to obtain the following shared key. But instead of reducing one party's public key to an element of different subgroups, there is another clever attack called a small subgroup attack that creates the different subgroup generators directly and sends them to one peer successively to obtain its private key. It is an active attack that doesn't work against ephemeral protocols that renew the Diffie-Hellman public key for every new key exchange. This is, for example, the case with TLS when using ephemeral Diffie-Hellman (DHE) as a key exchange during the handshake.

The attack is straight forward and summed up below:

1. Determine the prime factorization of the order of the group

$$\varphi(p) = \prod p_i^{k_i}$$

2. Find a generator for every subgroup of order $p_i^{k_i}$, this can be done by picking a random element α and computing

$$\alpha^{\varphi(p)/p_i^{k_i}} \pmod{p}$$

3. Send generators one by one as your public keys in different Diffie-Hellman key exchanges

4. Determine the value of x modulo $p_i^{k_i}$ for each shared key computed

[4] http://www.csee.umbc.edu/~lomonaco/s08/441/handouts/GarnerAlg.pdf

5. Recompute $x \pmod{\varphi(p)}$ with the CRT

The fourth step can be done by having access to an oracle telling you what the shared key computed by the victim is. In TLS this is done by brute-forcing the possible solutions and seeing which one has been used by the victim in their following encrypted messages (for example the MAC computation in the Finish message during the handshake). With these constraints the attack would be weaker than Pohlig-Hellman since the brute-force is slower than *Pollard Rho*, or even trial multiplication. Because of the previously stated limitations and the fact that this attack only works for rather small subgroups, we won't use it in this work.

3 A First Backdoor Attempt in Prime Groups

The naive approach to creating a backdoor would be to weaken the parameters enough to make the computation of discrete logarithms affordable. Making the modulus a prime of a special form ($r^e + s$ with small r and s) would facilitate the Special Number Field Sieve (SNFS) algorithm. Having a small modulus would also allow for easier pre-computation of the General Number Field Sieve (GNFS) algorithm. It is believed[ABD+15] that the NSA has enough power to achieve the first pre-computing phases of GNFS on 1024-bit primes, which would then allow them to compute discrete logarithms in such large groups in the matter of seconds.

But these ideas are pure computational advantages that involve no secret key to make the use of efficient backdoors possible. Moreover they are downright not practical: the attacker would have to re-do the pre-computing phase entirely for every different modulus, and the next generation of recommended modulus bitsize (2048+) would make these kind of computational advantages fruitless.

Another approach could be to use a generator of a smaller subgroup (without publishing what smaller subgroup we use) so that algorithms like *Pollard Rho* would be cost-effective again.

$$\begin{tikzcd}
\varphi(p) = p - 1 = \boxed{p_1} \times \cdots \times p_k \\
y = g^x \pmod{p} \arrow[u, "\text{order}"']
\end{tikzcd}$$

But then algorithms like *Pollard Kangaroo* that run in the same amount of time as *Pollard Rho* and that do not require the knowledge of the base's order could be used as well by anyone willing to try. This makes it a poorly hidden backdoor that we cannot qualify as NOBUS.

Our first contribution (CM-HSS) in section 4 makes both of these ideas possible by using a composite modulus. GNFS and SNFS can then be used modulo the factors of the composite modulus, or better as we will see, the generator's "small" subgroups can be concealed modulo the factors.

Back to our prime modulus. A second idea would be to set the scene for the Pohlig-Hellman algorithm to work. This can be done by fixing a prime modulus p such that $p - 1$ is B-smooth with B small enough for discrete logarithms in bases of order B to be possible.

$$\begin{tikzcd}
 & y = g^x \pmod{p} \\
 & \varphi(p) = p - 1 = p_1 \times \cdots \times p_k \arrow[dl, bend right] \arrow[dr, bend left] \arrow[d, phantom, "\text{Pohlig-Hellman}"] & \\
x \pmod{p_1} \arrow[dr] & \ldots \arrow[d, phantom, "\text{CRT}"] & x \pmod{p_k} \arrow[dl] \\
 & x \pmod{\varphi(p)} &
\end{tikzcd}$$

But this design is flawed in the same ways as the previous ones were: anyone can compute the order of the group (by subtracting 1 from p) and try to factor it. Choosing p such that $p - 1$ would include factors small enough to use one of the $\mathcal{O}(\sqrt{p})$ would make it dangerously factorisable. Using the *Elliptic Curve Method* (ECM), a factorization algorithm which complexity only depends on the size of the smallest factor (or for a full factorization, on the size of the second largest factor), the latest records[5] were able to find factors of around 300 bits. This necessary lower bound on the factors makes it unfeasible to use any of the $\mathcal{O}(\sqrt{p})$ algorithms

[5] http://www.loria.fr/~zimmerma/records/top50.html

that would take, for example, more than 2^{150} operations to solve the discrete logarithm of 300-bit orders.

Our second contribution in section 5 uses a composite modulus to hide the smoothness of the order (CM-HSO) as long as the modulus cannot be factored. This method is preferred from the first contribution as it might only need a change of modulus. For example, in many DH parameters or implementations, $g = 2$ as a generator is often used. While our first contribution will not allow any easy ways to find a specific generator, our second method will.

4 A Composite Modulus for a NOBUS Backdoor with a Hidden Subgroup (CM-HSS)

Our first NOBUS backdoor gets around the previous problems using a composite modulus $n = pq$ with p and q large enough to avoid the factorization of n. This requires the same precautions used to secure RSA instances, with n typically reaching 2048 bits and with two factors p and q nearing the same size.

With the factorization of n known, the discrete logarithm problem can be reduced modulo p and q and solved there, before being reconstructed modulo $\varphi(n)$ with the help of the CRT theorem.

$$\begin{tikzcd}
 & y = g^x \pmod{n = pq} \arrow[dl] \arrow[dr] & \\
y \pmod{p} \arrow[d, "\text{NFS/SNFS}"'] & & y \pmod{q} \arrow[d, "\text{NFS/SNFS}"] \\
x \pmod{p-1} \arrow[dr] & \text{CRT} & x \pmod{q-1} \arrow[dl] \\
 & x \pmod{(p-1)(q-1)} &
\end{tikzcd}$$

p and q could be hand-picked as SNFS primes, or we could use GNFS to compute the discrete logarithm modulo p and q. But a more efficient way exists to ease the discrete logarithm problem. Choosing a generator g such that both g modulo p and g modulo q generate "small" subgroups, would allow us to compute two discrete logarithms in two small subgroups instead of one discrete logarithm in one large group.

For example, we could pick p and q such that $p - 1 = 2p_1p_2$ and $q - 1 = 2q_1q_2$ with p_1 and q_1 two small prime factors and p_2, q_2 two large prime factors. Lagrange's theorem tells us that the possible orders of the subgroups are divisors of the group order. This mean we can probably find an element g of order p_1q_1 to be our Diffie-Hellman generator.

$$\begin{tikzcd}
 & p-1 \arrow[dl] \arrow[d] \arrow[dr] & & & q-1 \arrow[dl] \arrow[d] \arrow[dr] & \\
2 & \boxed{p_1} & p_2 & 2 & \boxed{q_1} & q_2 \\
 & g \pmod{p} \arrow[u, "\text{subgroup of order}"'] & & & g \pmod{q} \arrow[u] &
\end{tikzcd}$$

By reducing the discrete logarithm problem $y = g^x$ modulo p and q with our new backdoored generator, we can compute x modulo $p-1$ and $q-1$ more easily and then recompute an equivalent secret key modulo $(p-1)(q-1)$. This will find the exact original secret key with a probability of $\frac{1}{4p_2q_2}$, which is tiny, but this doesn't matter since the shared key we will compute with that solution and the other peer's public key will be a valid shared key.

Proof. Let $a + k_ap_1q_1$ be Alice's public key for $k_a \in \mathbb{Z}$ and let $b + k_bp_1q_1$ be Bob's public key for $k_b \in \mathbb{Z}$, then Bob's shared key will be $(g^{a+k_ap_1q_1})^{b+k_bp_1q_1} = g^{ab} \pmod{n}$.

Let $a + k_c p_1 q_1$ be the solution we found for $k_c \in \mathbb{Z}$,
then the shared key we will compute will be $(g^{b+k_b p_1 q_1})^{a+k_c p_1 q_1} = g^{ab} \pmod{n}$, which is the same as Bob's shared key. $\square$

We used the *Pollard Rho* function in Sage 6.10 on a Macbook Pro with an i7 Intel Core @ 3.1GHz to compute discrete logarithms modulo safe primes of diverse bitsizes. The results are summed up in the table below.

order size	expected complexity	time
40 bits	2^{20}	01s
45 bits	2^{22}	04s
50 bits	2^{25}	34s

A stronger and more clever attacker would parallelize this algorithm on more powerful machines to obtain better numbers. To be able to exploit the backdoor "live" we want a running-time close to zero. Using an 80-bit integer as our generator's order, someone with no knowledge of the factorization of the modulus would take around 2^{40} operations to compute a discrete logarithm while this would take us on average 2^{21} thanks to the trapdoor. A more serious adversary with a higher computation power and a care for security might want to choose a 200-bit integer as the generator's order. For that they would need to be able to perform 2^{50} operations instantaneously if they would want to tamper with the encrypted communications following the key exchange, while an outsider would have to perform an "impossible" number of 2^{100} operations. The size of the two primes p and q, and of the resulting $n = pq$, should be chosen large enough to resist against the same attacks as RSA. That is an n of 2048 bits with p and q both being 1024 bits long would suffice.

To use such a backdoor, one must not only generate two primes p and q to satisfy the previous shape, but also find a specific generator g. This is not a hard task, unless you want to use a specific generator g. For example many libraries use $g = 2$ by default, implementing this backdoor would mean changing both the modulus and the generator. This is because the probability that an element in a group of order q is the generator of a subgroup of order d is $\frac{d}{q}$. This means that with our example $g = 2$, we would need to generate many modulus hoping that $g = 2$ as a generator would work. The probability that it would work for each try would be:

$$\frac{p_1 p_2}{(p-1)(q-1)} \sim \frac{1}{pq} = \frac{1}{n}$$

This is obviously too small of a probability for us to try to generate many parameters until one admits our targeted g as a generator of our "small" subgroup. This is a problem if we want to only replace the modulus of an implementation to activate our backdoor. Since changing only one value would be more subtle than changing two values, our next contribution revises the way we generate the backdoor parameters to solve this problem.

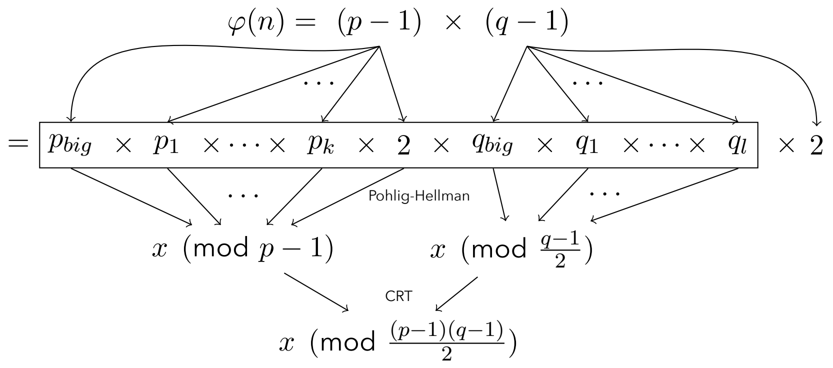
5 A Composite Modulus for a NOBUS Backdoor with a B-Smooth Order (CM-HSO)

Let's start again with a composite modulus $n = pq$, but this time let's choose p and q such that $p-1$ and $q-1$ are both B-smooth with B small enough for the discrete logarithm to be doable in subgroups of order B. We'll see later how to choose B.

Let $p-1 = p_1 \times \cdots \times p_k \times 2$ and $q-1 = q_1 \times \cdots \times q_l \times 2$ such that $lcm(p-1, q-1) = 2$ and such that $p_i \leq B$ and $q_j \leq B$ for all $i \in [\![1,k]\!]$ and $j \in [\![1,l]\!]$ respectively. This makes the order of the group $\varphi(n) = (p-1)(q-1)$ B-smooth.

Constructing the Diffie-Hellman modulus this way permits anyone with both the knowledge of the order factorization and the ability of computing the discrete logarithm in subgroups of order B, to compute the discrete logarithm modulo n by using the Pohlig-Hellman method.

Since $p-1$ and $q-1$ are both B-smooth, they are susceptible to be factored with the *Pollard's p-1* factorization algorithm, a factorization algorithm that can find a factor p of n if $p-1$ is partially-smooth. RSA counters this problem using safe primes of the form $p = q+1$ with q prime as well, but this would break our backdoor. Instead, as a way of countering *Pollard's p-1* we can add a large factor to both $p-1$ and $q-1$ that we will call p_{big} and q_{big} respectively.


$\varphi(n) = (p-1) \times (q-1)$
$= \boxed{p_{big} \times p_1 \times \cdots \times p_k \times 2 \times q_{big} \times q_1 \times \cdots \times q_l} \times 2$
Pohlig-Hellman
$x \pmod{p-1}$
$x \pmod{\frac{q-1}{2}}$
CRT
$x \pmod{\frac{(p-1)(q-1)}{2}}$

To exploit this backdoor we can reduce our public key y modulo p and q, as we did in our first method, and proceed with Pohlig-Hellman's algorithm there. This is not a necessary step but this will reduce the size of the numbers in our calculations, speeding up the attack. We then carry on with CRT to recompute the private key modulo its order, which can be picked at a secure maximum of $\frac{(p-1)(q-1)}{2}$, which brings around the same security promises of a safe-prime modulus. This is because of the following isomorphy we have: $(\mathbb{Z}_n)^* \simeq (\mathbb{Z}_p)^* \times (\mathbb{Z}_q)^*$, with the product's orders $s = |(\mathbb{Z}_p)^*|$ and $t = |(\mathbb{Z}_q)^*|$ not being coprimes ($gcd(p-1, q-1) = 2$). This results in a non-cyclic group with an upper-bound on possible subgroup orders of $lcm(s,t) = \frac{(p-1)(q-1)}{2}$.

To decide how big p_{big} and q_{big} should be, we can look at the world's records for the *Pollard's p-1* factorization algorithm,[6] the largest B2 parameter (the large factor) used in a factorization is $10^{15} \sim 50bits$ in 2015. As with our previous method, we could use much larger factors of around 100 bits to avoid any powerful adversaries and have an agreeable 2^{51} computations on average to solve the discrete logarithm problem in these large subgroups.

While the previous method gave us a quadratic edge over someone unknowledgeable of the factorization

[6] http://www.loria.fr/~zimmerma/records/Pminus1.html

of n, this new method rises the security of our overall scheme to the one of a perfectly secure Diffie-Hellman use. Its security also relies on the RSA's assumption that factoring n is difficult if n is large enough. More than that, the probability of having a targeted element be a valid generator can be as large as $\frac{1}{2}$ in our example of a secure subgroup of order $\frac{(p-1)(q-1)}{2}$. This will allow us to easily generate a backdoored modulus that will fit a specific generator, thus increasing the stealthiness of the implementation phase of our scheme.

In the case where the large two subgroups of order p_{big} and q_{big} need to be avoided, one could think about trying to generate many modulus hoping that the targeted g would fit. This can be done with probability $\frac{1}{2p_{big}q_{big}}$ for each try, which is way too low. But worse, this would give a free start to someone trying to factor the modulus using *Pollard's p-1*. Another way would be to pass a fake order to the program, forcing it to generate small ephemeral private keys upper-bounded by $\frac{\varphi(n)}{p_{big}q_{big}}$. After this, proceeding to use Pohlig-Hellman over the small factors, ignoring p_{big} and q_{big}, would be enough to find the private key. This can be done in OpenSSL or libraries making use of it by passing the fake order to OpenSSL via *dh->q*. Of course doing this would bring back our first method's issues by having to add extra lines of code to our malicious patch.

6 Implementing and Exploiting the Backdoor in TLS

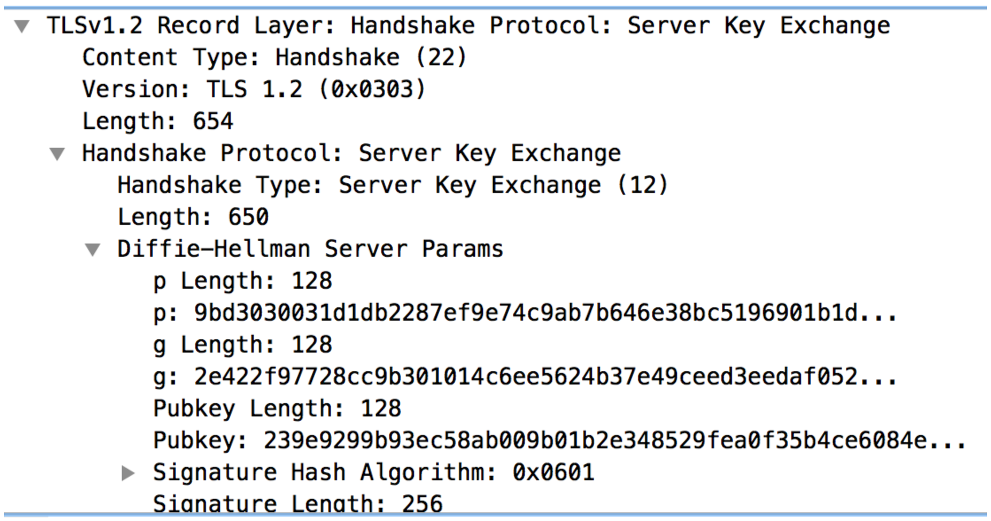
Theoretically, any application including Diffie-Hellman might be backdoored using one of the previous two methods. As TLS is one of the most well known protocols using Diffie-Hellman it is particularly interesting to abuse for a field test of our work.

Most TLS applications making use of the Diffie-Hellman algorithm for the handshake—although this is an algorithm rarely used in TLS—would have their DH public key and parameters baked into user's or server's generated certificates. Interestingly, the parameters of the, much more commonly used, *ephemeral* version of Diffie-Hellman used to add the property of *Perfect Forward Secrecy*, are rarely chosen by end users and thus never engraved into user's or server's certificates. Furthermore, most libraries implementing the TLS protocol (socat, Apache, NGINX, ...) have predefined or hardcoded ephemeral DH parameters. Developers using those libraries will rarely generate their own parameters and will use the default ones. This was the source of many discussions after being pointed out by *Logjam*[ABD+15] last year, creating a movement of awareness, pushing people to migrate to bigger parameters and increase the bitsizes of applications' Diffie-Hellman modulus from 1024 or lower to 2048+ bits. This trend seems like the perfect excuse to submit a backdoored patch claiming to improve the security of a library. We'll first see how TLS works with ephemeral Diffie-Hellman in the next section. Followed will be a demonstration on how the backdoor was implemented in real open source libraries. Finally we'll explain how our setup worked to make use of the backdoor.

6.1 Background

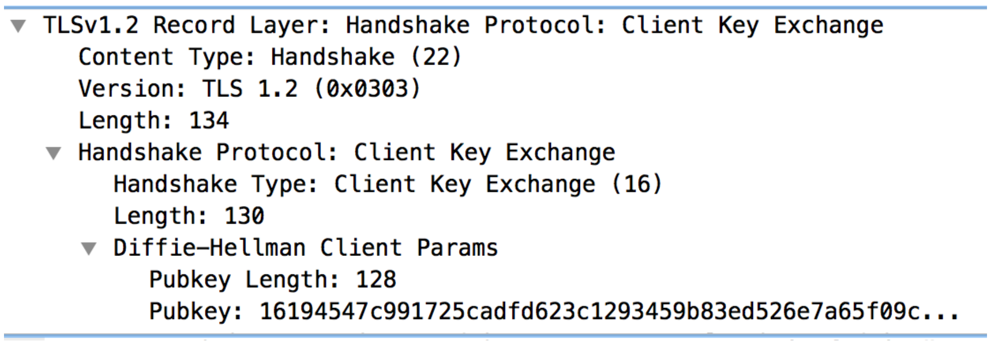
An ephemeral handshake allows two parties to negotiate a "fresh" set of keys for every new TLS connection. This has become the preferred way of using TLS as it increases its security, providing the property that we call *Forward Secrecy* or *Perfect Forward Secrecy*, that is: if the long term key is compromised, recorded past communications won't be affected and future communications will still resist passive attacks. This is done by using one of the two Diffie-Hellman algorithms provided by TLS: "normal" Diffie-Hellman present in the ciphersuites containing *DHE* in their names and Elliptic Curve Diffie-Hellman (ECDH) present in the ciphersuites containing *ECDHE* in their names. Note that the concept of "*ephemeral*" is not defined the same by everyone: the default behavior of OpenSSL, up until recent versions, has been to generate the ephemeral DH key at boot time and cache it until reboot, unless specified not to do so. Such behavior would greatly speed up our attack.

At the start of a new *ephemeral handshake*, both the server and the client will send each other their ephemeral DH (DHE) public keys via a *ServerKeyExchange* and a *ClientKeyExchange* message respectively. The server will dictate as well what the DHE parameters are via the same *ServerKeyExchange* message.



```
▼ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 654
  ▼ Handshake Protocol: Server Key Exchange
      Handshake Type: Server Key Exchange (12)
      Length: 650
    ▼ Diffie-Hellman Server Params
        p Length: 128
        p: 9bd3030031d1db2287ef9e74c9ab7b646e38bc5196901b1d...
        g Length: 128
        g: 2e422f97728cc9b301014c6ee5624b37e49ceed3eedaf052...
        Pubkey Length: 128
        Pubkey: 239e9299b93ec58ab009b01b2e348529fea0f35b4ce6084e...
      ▶ Signature Hash Algorithm: 0x0601
        Signature Length: 256
```

Figure 3: The serverKeyExchange message



```
▼ TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 134
  ▼ Handshake Protocol: Client Key Exchange
      Handshake Type: Client Key Exchange (16)
      Length: 130
    ▼ Diffie-Hellman Client Params
        Pubkey Length: 128
        Pubkey: 16194547c991725cadfd623c1293459b83ed526e7a65f09c...
```

Figure 4: The clientKeyExchange message

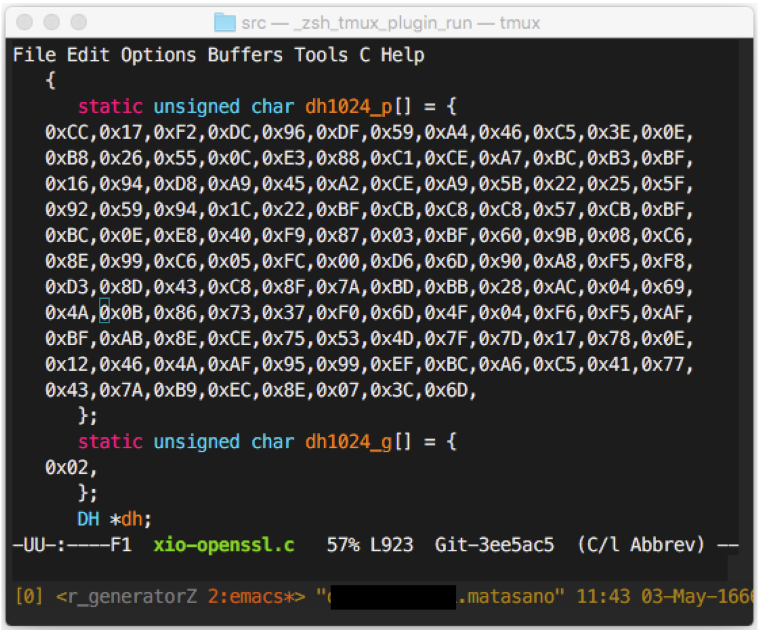
Let **c** and **s** be the client and the server public keys respectively. The following computation is done on each side, right after the key exchange, to derive the session keys that will encrypt further communications (including final handshake messages):

1. premaster_secret $= g^{cs} \pmod{n}$
2. master_secret = PRF(premaster_secret, "master secret", ClientHello.random + ServerHello.random)
3. keys = PRF(master_secret, "key expansion", ServerHello.random + ClientHello.random)

Right after trading their ephemeral DH public keys, the TLS peers compute the Diffie-Hellman algorithm by exponentiating the other's public key with their own private key. The output is stored in a *premaster_secret* variable that is sent into a pseudo-random function (PRF) with the string "master secret" and the public values *random* of both parties taken from their Hello message as parameters. This is because the DH output can be of fluctuating lengths: TLS offers several parameters and algorithms to perform this part of the handshake, passing it through a PRF first aims to normalize its size before deriving the keys from it. The output of that first PRF is then sent into the same PRF repeatedly along with different arguments: the string "key expansion" and the reversed order of the *random* values we just used, until enough bits are produced for the many keys used to encrypt and authenticate the post-handshake communications.

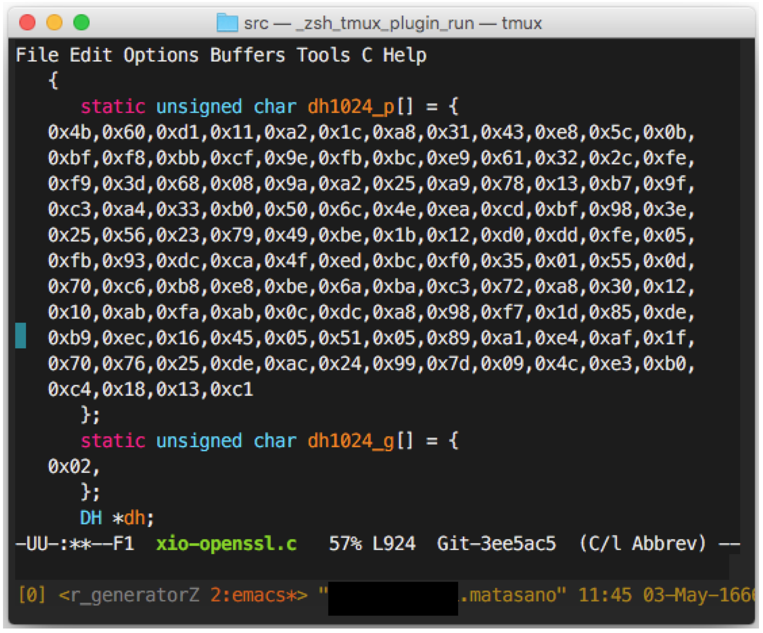
Two authentication keys are first derived, *client_write_MAC_key* and *server_write_MAC_key*, one for each direction. Then two encryption keys are derived as well, *client_write_key* and *server_write_key*. For AEAD ciphers, MAC keys are ignored and two more values after the encryption keys are derived: *client_write_IV* and *server_write_IV*.

6.2 Implementation



```
src — _zsh_tmux_plugin_run — tmux
File Edit Options Buffers Tools C Help
    {
        static unsigned char dh1024_p[] = {
    0xCC,0x17,0xF2,0xDC,0x96,0xDF,0x59,0xA4,0x46,0xC5,0x3E,0x0E,
    0xB8,0x26,0x55,0x0C,0xE3,0x88,0xC1,0xCE,0xA7,0xBC,0xB3,0xBF,
    0x16,0x94,0xD8,0xA9,0x45,0xA2,0xCE,0xA9,0x5B,0x22,0x25,0x5F,
    0x92,0x59,0x94,0x1C,0x22,0xBF,0xCB,0xC8,0xC8,0x57,0xCB,0xBF,
    0xBC,0x0E,0xE8,0x40,0xF9,0x87,0x03,0xBF,0x60,0x9B,0x08,0xC6,
    0x8E,0x99,0xC6,0x05,0xFC,0x00,0xD6,0x6D,0x90,0xA8,0xF5,0xF8,
    0xD3,0x8D,0x43,0xC8,0x8F,0x7A,0xBD,0xBB,0x28,0xAC,0x04,0x69,
    0x4A,0x0B,0x86,0x73,0x37,0xF0,0x6D,0x4F,0x04,0xF6,0xF5,0xAF,
    0xBF,0xAB,0x8E,0xCE,0x75,0x53,0x4D,0x7F,0x7D,0x17,0x78,0x0E,
    0x12,0x46,0x4A,0xAF,0x95,0x99,0xEF,0xBC,0xA6,0xC5,0x41,0x77,
    0x43,0x7A,0xB9,0xEC,0x8E,0x07,0x3C,0x6D,
        };
        static unsigned char dh1024_g[] = {
    0x02,
        };
        DH *dh;
-UU-:----F1  xio-openssl.c   57% L923  Git-3ee5ac5  (C/l Abbrev) --

[0] <r_generatorZ 2:emacs*> "[redacted].matasano" 11:43 03-May-166
```

Figure 5: socat's *xio-openssl.c* file



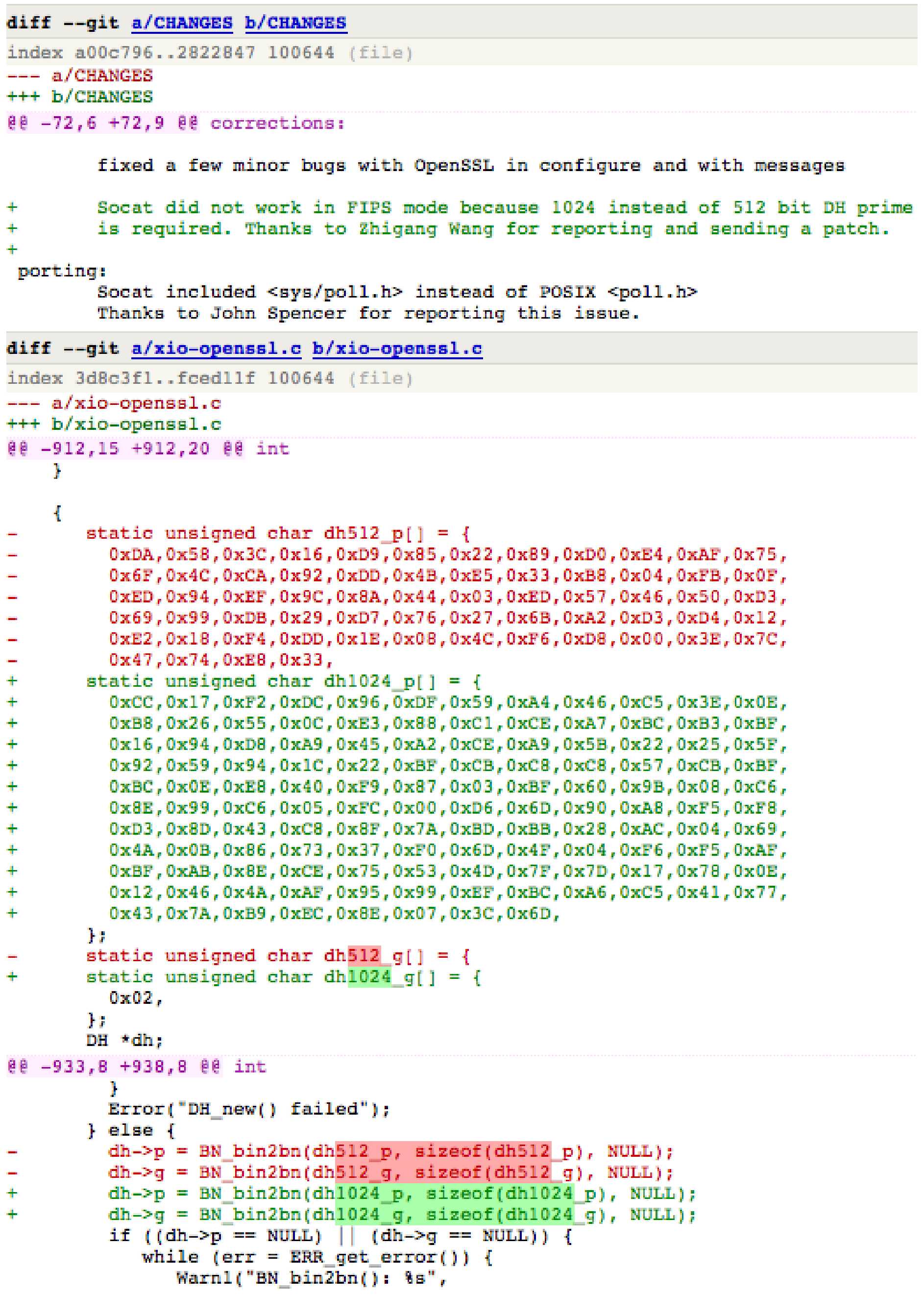
```
src — _zsh_tmux_plugin_run — tmux
File Edit Options Buffers Tools C Help
    {
        static unsigned char dh1024_p[] = {
    0x4b,0x60,0xd1,0x11,0xa2,0x1c,0xa8,0x31,0x43,0xe8,0x5c,0x0b,
    0xbf,0xf8,0xbb,0xcf,0x9e,0xfb,0xbc,0xe9,0x61,0x32,0x2c,0xfe,
    0xf9,0x3d,0x68,0x08,0x9a,0xa2,0x25,0xa9,0x78,0x13,0xb7,0x9f,
    0xc3,0xa4,0x33,0xb0,0x50,0x6c,0x4e,0xea,0xcd,0xbf,0x98,0x3e,
    0x25,0x56,0x23,0x79,0x49,0xbe,0x1b,0x12,0xd0,0xdd,0xfe,0x05,
    0xfb,0x93,0xdc,0xca,0x4f,0xed,0xbc,0xf0,0x35,0x01,0x55,0x0d,
    0x70,0xc6,0xb8,0xe8,0xbe,0x6a,0xba,0xc3,0x72,0xa8,0x30,0x12,
    0x10,0xab,0xfa,0xab,0x0c,0xdc,0xa8,0x98,0xf7,0x1d,0x85,0xde,
    0xb9,0xec,0x16,0x45,0x05,0x51,0x05,0x89,0xa1,0xe4,0xaf,0x1f,
    0x70,0x76,0x25,0xde,0xac,0x24,0x99,0x7d,0x09,0x4c,0xe3,0xb0,
    0xc4,0x18,0x13,0xc1
        };
        static unsigned char dh1024_g[] = {
    0x02,
        };
        DH *dh;
-UU-:**--F1  xio-openssl.c   57% L924  Git-3ee5ac5  (C/l Abbrev) --

[0] <r_generatorZ 2:emacs*> "[redacted].matasano" 11:45 03-May-166
```

Figure 6: The same backdoored socat file after our changes

socat is a useful command line tool that makes use of OpenSSL underneath the surface, so it is what we first used to quickly test our backdoor parameters. With the help of the *dhparam* argument, we can use an

ASN.1 file encoded in the DER format to specify our backdoored DH parameters to the program. But as this exercise is one of implementation, we will show how we modified the source code of socat to introduce the backdoor to the masses. We started by generating small parameters with the second method (CM-HSO) for our tests. Since socat's generator is 2, we only had to modify the modulus in the *dh1024_p* variable of the *xio-openssl.c* file. Above are the differences between a valid version and a backdoored version of socat.

If we want to submit such a patch as an excuse to increase the size of the DH parameters, we could mimic socat's case[7]:



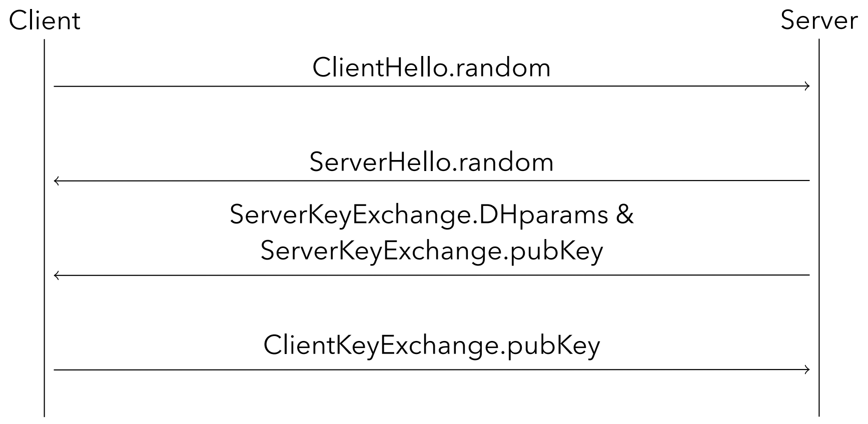
```
diff --git a/CHANGES b/CHANGES
index a00c796..2822847 100644 (file)
--- a/CHANGES
+++ b/CHANGES
@@ -72,6 +72,9 @@ corrections:

        fixed a few minor bugs with OpenSSL in configure and with messages

+       Socat did not work in FIPS mode because 1024 instead of 512 bit DH prime
+       is required. Thanks to Zhigang Wang for reporting and sending a patch.
+
 porting:
        Socat included <sys/poll.h> instead of POSIX <poll.h>
        Thanks to John Spencer for reporting this issue.
diff --git a/xio-openssl.c b/xio-openssl.c
index 3d8c3f1..fced11f 100644 (file)
--- a/xio-openssl.c
+++ b/xio-openssl.c
@@ -912,15 +912,20 @@ int
    }

    {
-      static unsigned char dh512_p[] = {
-        0xDA,0x58,0x3C,0x16,0xD9,0x85,0x22,0x89,0xD0,0xE4,0xAF,0x75,
-        0x6F,0x4C,0xCA,0x92,0xDD,0x4B,0xE5,0x33,0xB8,0x04,0xFB,0x0F,
-        0xED,0x94,0xEF,0x9C,0x8A,0x44,0x03,0xED,0x57,0x46,0x50,0xD3,
-        0x69,0x99,0xDB,0x29,0xD7,0x76,0x27,0x6B,0xA2,0xD3,0xD4,0x12,
-        0xE2,0x18,0xF4,0xDD,0x1E,0x08,0x4C,0xF6,0xD8,0x00,0x3E,0x7C,
-        0x47,0x74,0xE8,0x33,
+      static unsigned char dh1024_p[] = {
+        0xCC,0x17,0xF2,0xDC,0x96,0xDF,0x59,0xA4,0x46,0xC5,0x3E,0x0E,
+        0xB8,0x26,0x55,0x0C,0xE3,0x88,0xC1,0xCE,0xA7,0xBC,0xB3,0xBF,
+        0x16,0x94,0xD8,0xA9,0x45,0xA2,0xCE,0xA9,0x5B,0x22,0x25,0x5F,
+        0x92,0x59,0x94,0x1C,0x22,0xBF,0xCB,0xC8,0xC8,0x57,0xCB,0xBF,
+        0xBC,0x0E,0xE8,0x40,0xF9,0x87,0x03,0xBF,0x60,0x9B,0x08,0xC6,
+        0x8E,0x99,0xC6,0x05,0xFC,0x00,0xD6,0x6D,0x90,0xA8,0xF5,0xF8,
+        0xD3,0x8D,0x43,0xC8,0x8F,0x7A,0xBD,0xBB,0x28,0xAC,0x04,0x69,
+        0x4A,0x0B,0x86,0x73,0x37,0xF0,0x6D,0x4F,0x04,0xF6,0xF5,0xAF,
+        0xBF,0xAB,0x8E,0xCE,0x75,0x53,0x4D,0x7F,0x7D,0x17,0x78,0x0E,
+        0x12,0x46,0x4A,0xAF,0x95,0x99,0xEF,0xBC,0xA6,0xC5,0x41,0x77,
+        0x43,0x7A,0xB9,0xEC,0x8E,0x07,0x3C,0x6D,
       };
-      static unsigned char dh512_g[] = {
+      static unsigned char dh1024_g[] = {
         0x02,
       };
       DH *dh;
@@ -933,8 +938,8 @@ int
         }
         Error("DH_new() failed");
       } else {
-        dh->p = BN_bin2bn(dh512_p, sizeof(dh512_p), NULL);
-        dh->g = BN_bin2bn(dh512_g, sizeof(dh512_g), NULL);
+        dh->p = BN_bin2bn(dh1024_p, sizeof(dh1024_p), NULL);
+        dh->g = BN_bin2bn(dh1024_g, sizeof(dh1024_g), NULL);
         if ((dh->p == NULL) || (dh->g == NULL)) {
            while (err = ERR_get_error()) {
               Warn1("BN_bin2bn(): %s",
```

Figure 7: The commit diff page of the socat "presumed" backdoor

[7] http://repo.or.cz/socat.git/commitdiff/281d1bd6515c2f0f8984fc168fb3d3b91c20bdc0

6.3 Exploitation

To exploit this kind of backdoor, we first need to obtain a Man-In-The-Middle position between the client and the server. This can be done passively by obtaining posterior access to logs of TLS records, but this was done actively in our proof of concept[8] by using a machine as a proxy to the server and making the client connect to the proxy directly instead of the server. The proxy unintelligently forwards the packets back and forth until a TLS connection is initiated, it then observes the handshake, storing the *random* values at first, until the server decides to send its public Diffie-Hellman parameters to be used in an *ephemeral* key exchange. If the proxy recognizes the backdoor parameters in the server's *ServerKeyExchange* message, it runs the attack, recovering one party's private key and computing the session keys out of that information. With the session keys in hand, the proxy can then observe the traffic in clear and even tamper with the messages being exchanged.



Depending on the security margins chosen during the generation of the backdoor, and on the computing power of the attacker, it may be the case that the attacker would not be able to derive the session keys until the first few messages have been exchanged, exempting them from tampering. For better results, the work could be parallelized and the two public keys could be attacked simultaneously as one might be recovered more quickly than the other. As soon as the private key of one party is recovered, the Diffie-Hellman and the session keys computations are done in a negligible time, and the proxy can start live decrypting and live tampering with the packets. If the attacker really wants to be able to tamper with the first messages, it can delay the end of the handshake by sending *TLS warning alerts* that can keep a handshake alive indefinitely or for a period of time depending on the TLS implementation used by both parties.

[8] https://github.com/mimoo/Diffie-Hellman_Backdoor

7 Detecting a Backdoor and Defending Against One

In the course of this work several open source libraries were tested for composite modulus with no positive results. TLS handshakes of the full range of IPv4 addresses obtained from scans.io were inspected on March 3rd, 2016. A total of 50,222,805 handshakes were analyzed from which 4,522,263 were augmented with the use of ephemeral Diffie-Hellman. From these numbers, only 30 handshakes used a composite modulus, all of them had a small factor but none of them could be factored in less than 5 hours using the ECM or *Pollard's p-1* factorization algorithms. Most IPs were hosting webpages, in some cases the same one. All administrators were contacted about the issue.

Our contributions should withstand any kind of reversing and thus we should not be able to detect any backdoor produced by people who would have reached the same conclusions as ours. The addition of easy to find small factors could have been intentionally done to provide plausible deniability. Interestingly, it is also hard to differentiate a mistake in the modulus generation from a backdoor. From the Handbook of Applied Cryptography,[9] fact 3.7:

Definition 1. *Let n be chosen uniformly at random from the interval $[1, x]$.*

1. *if $1/2 \leq \alpha \leq 1$, then the probability that the largest prime factor of n is $\leq x^{\alpha}$ is approximately $1 + ln(\alpha)$. Thus, for example, the probability than n has a prime factor $> \sqrt(x)$ is $ln(2) \approx 0.69$*
2. *The probability that the second-largest prime factor of n is $\leq x^{0.2117}$ is about $1/2$.*
3. *The expected total number of prime factors of n is $lnlnx + \mathcal{O}(1)$. (If $n = \prod p_i^{e_i}$, the total number of prime factors of n is $\sum e_i$.)*

Since it might be easier to visualize this with numbers:

1. A 1024-bit composite modulus n probability to have a prime factor greater than 512 bits is ≈ 0.69.
2. The probability that the second-largest prime factor of n is smaller than 217 bits is $1/2$.
3. The total number of prime factors of n is expected to be 7.

Considering that a full factorization with ECM runs in a complexity tied to the size of the second largest factor, it might be hard or impossible to do it half of the time. The rest of the time it might take a bit of work, but since the largest factor found using ECM[10] is 274 bits, it is possible.

The question of how to avoid these kinds of backdoors or weaknesses is also interesting and well understood, but rarely done correctly. First, it is known that by using safe primes–primes of the form $2q + 1$ with q prime–the generator's subgroup will have an order close to the modulus' size. Since it is easy to check if a number is a safe prime, the client should also only accept such moduli. The current state is that most programs don't even check for prime modulus. As an example, no browser currently warns the user if a composite modulus is detected.

Another way to prevent this is to have a pre-defined list of public parameters[LK15, Kiv15, Gil15], this would make Diffie-Hellman look similar to Elliptic Curve Diffie-Hellman in the sense that only a few curves are pre-defined and accepted in most exchanges.

Both mitigations can be hard to integrate if the two endpoints of a key exchange are not controlled. For example, this is the case between browsers' and websites' TLS connections where the browser is a different

[9] http://cacr.uwaterloo.ca/hac/about/chap3.pdf
[10] http://www.loria.fr/~zimmerma/records/top50.html

program from what is running on the server. Asserting for these special primes might just break the connection, which would be worse from the user's perspective. This is why Google Chrome is currently removing DHE from its list of supported cipher suites,[11] and recommending server administrators to migrate from DHE to ECDHE. This is also one of the recommendations from Logjam[ABD$^+$15]. These security measures might very well prevent the attacks described in this work. Backdooring ECDHE in a stealthy way as we did with DHE remains an open problem.

[11] https://groups.google.com/a/chromium.org/forum/m/#!topic/blink-dev/AAdv838-koo/discussion

8 Conclusion

Many cryptographic constructions are not subject to change, unless a breakthrough comes along and the whole construction has to be replaced. Very rarely the excuse of updating a reviewed and considered strong cryptographic implementation to change a single number comes along, and very few people understand such subtle changes. According to the grading system of a whitepaper by Schneier et al[SFKR15], here is how such a backdoor scores:

- ***Medium*** *undetectability*: To discover the backdoor one would have to test for the primality of the modulus. A pretty easy task, although not typically performed as seen with the socat's case where it took them more than a year to realize the composite modulus.
- ***High*** *lack of conspiracy*: In the case of socat only the person who had submitted the vulnerability would be the target of investigation. It turns out he is a regular employee at Oracle.
- ***High*** *plausible deniability*: Three things help us in the creation of a good story in the socat's case: reversing bytes of the fake prime gives us a prime, some small factors were found, anyone with weak knowledge of cryptography could have submitted a composite number.
- ***Medium*** *ease of use*: Man-in-the-middling the attack and observing the first handshake would allow the attacker to take advantage of the backdoor.
- ***High*** *severity*: Having access to that backdoor lets us observe or, if exploited in real time, tamper with any communications made over TLS.
- ***Medium*** *durability*: System admins would have to update to newer versions to remove the backdoor.
- ***High*** *monitorability*: The saboteur cannot detect if other attackers are taking advantage of the backdoor, which is OK since the backdoor in this work are NOBUS ones.
- ***High*** *scale*: Backdooring an open-source library would allow access to many systems' and users' communications.
- ***High*** *precision*: The saboteur doesn't weaken any system, only the saboteur himself can access the backdoor.
- ***High*** *control*: Like Dual EC, only the saboteur can exploit the backdoor.

While this work is mostly a fictive exercise, we hope to raise awareness in the need for better toolings and deeper reviews of open source, as well as closed source, implementations of cryptographic algorithms.

Acknowledgements

Many thanks to both Scott Fluhrer and Scott Contini who have been of precious help in the core ideas of this paper.

Thanks as well to Pete L. Clark, Tom Ritter, Mike Brown, Roman Zabicki, Vincent Lynch, Drew Suarez, Divya Natesan, Ryan Koppenhaver and Andy Grant for feedback and discussions.

References

[ABD$^+$15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*, October 2015. https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf. 4, 10, 16, 22

[BLN15] Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. Dual ec: A standardized back door. Cryptology ePrint Archive, Report 2015/767, 2015. http://eprint.iacr.org/2015/767. 4

[CCG$^+$16] Stephen Checkoway, Shaanan Cohney, Christina Garman, Matthew Green, Nadia Heninger, Jacob Maskiewicz, Eric Rescorla, Hovav Shacham, and Ralf-Philipp Weinmann. A systematic analysis of the juniper dual ec incident. Cryptology ePrint Archive, Report 2016/376, 2016. http://eprint.iacr.org/2016/376. 4

[Gil15] Daniel Kahn Gillmor. Ietf draft: Negotiated finite field diffie-hellman ephemeral parameters for tls. Internet-Draft draft-ietf-tls-negotiated-ff-dhe-10, Internet Engineering Task Force, 2015. https://tools.ietf.org/html/draft-ietf-tls-negotiated-ff-dhe-10. 21

[Kiv15] Tero Kivinen. Rfc 3526: More modular exponential (modp) diffie-hellman groups for internet key exchange (ike). RFC 3526, 2015. https://rfc-editor.org/rfc/rfc3526.txt. 21

[LK15] Matt Lepinski and Dr. Stephen T. Kent. Rfc 5114: Additional diffie-hellman groups for use with ietf standards. RFC 5114, 2015. https://rfc-editor.org/rfc/rfc5114.txt. 21

[PH78] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance, 1978. http://www-ee.stanford.edu/~hellman/publications/28.pdf. 6

[Res13] Eric Rescorla. Rfc 2631: Diffie-hellman key agreement method. RFC 2631, 2013. https://rfc-editor.org/rfc/rfc2631.txt. 4

[SF] Shinde and Fadewar. Faster rsa algorithm for decryption using chinese remainder theorem. http://www.techscience.com/doi/10.3970/icces.2008.005.255.pdf. 7

[SF07] Shumow and Ferguson. On the possibility of a back door in the nist sp800-90 dual ec prng, 2007. Crypto 2007. http://rump2007.cr.yp.to/15-shumow.pdf. 4

[SFKR15] Bruce Schneier, Matthew Fredrikson, Tadayoshi Kohno, and Thomas Ristenpart. Surreptitiously weakening cryptographic systems. Cryptology ePrint Archive, Report 2015/097, 2015. http://eprint.iacr.org/. 23

[Zuc13] Robert Zuccherato. Rfc 2785: Methods for avoiding the small-subgroup attacks on the diffie-hellman key agreement method for s/mime. RFC 2785, 2013. https://rfc-editor.org/rfc/rfc2785.txt. 4