How to Backdoor Diffie-Hellman

David Wong

NCC Group, March 2016

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's operating system ScreenOS and a non-prime modulus in the Socat open-source tool. The question on how fragile cryptographic constructions are if we do not use Nothing-Up-My-Sleeve numbers has came up in many papers, but the question on how to introduce a backdoor in an already secure, safe and easy to audit implementation has so far never been researched (in the public).
In this work we present two ways of building a Nobody-But-Us Diffie-Hellman backdoor with a composite modulus and a hidden subgroup (CMHS) and with a composite modulus and a smooth order (CMSO). We then explain how we subtly implemented it and exploited it in various open source libraries using the TLS protocol.

Keywords: Diffie-Hellman, Ephemeral, DHE, NOBUS, Backdoor, Discrete Logarithm, Small Subgroup Attack, Pohlig-Hellman, Pollard Rho, Factorization, Pollard's p-1, ECM, Dual-EC, Juniper, Socat

1 Introduction

Around Christmas 2015, a company named *Juniper* released an out of cycle security bulletin[1]. Two vulnerabilities were being semi-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 try solving this puzzle. Quickly, by diffing both the patched and vulnerable binaries, the two issues were pinpointed. While one of the vulnerability 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 was modified. More accurately, a number was replaced. The introduction of the vulnerability was so trivial that the simple use of the unix command line tool `strings` was enough to discover the change.

[1] https://kb.juniper.net/InfoCenter/index?page=contentid=JSA10713actp=search

Figure 1: The strings of the patched binary

Figure 2: The strings of the vulnerable binary

Behind that modified number was hiding a *Dual EC* value. Dual EC is a Pseudo-Random Number Generator (PRNG) that is believed to have been backdoored by the NSA[2] [1]. 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, leaving them the only ones able to climb back to the PRNG's internal state from random outputs, and then being able to predict the PRNG's future random outputs. But by modifying one of the point (Q), the change in Juniper's code magically allowed another organization, or person, to access that backdoor in place of the NSA.
Although the quest to find the Juniper's backdoor and the numerous open questions that arose from that work are a fascinating read by itself[3] [2], it is only the introduction of this work that aims to show how secure and strong cryptographic constructions are a simple and subtle change away from being your own secretive pipe-show.

On February the 1st 2016, only a few months after Juniper's debacle, *Socat* published a security advisory of its own[4]:

> 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 of Juniper's problem, a single number was changed. This time it was the public modulus number used to generate the ephemeral Diffie-Hellman keys of both parties during Socat's TLS handshakes. An algorithm that was contrarily to Dual-EC considered secure from the start, but badly understood as well.

[2] Bernstein, Lange and Niederhagen - Dual EC: A Standardized Back Door

[3] Checkoway et al - A Systematic Analysis of the Juniper Dual EC Incident (2016)

[4] http://www.openwall.com/lists/oss-security/2016/02/01/4

As the *Logjam*[5][3] paper had discovered earlier last year, most servers would use Diffie-Hellman to perform ephemeral handshakes, and generate parameters from hardcoded defaults. The paper launched a wave of discussion around how users should use Diffie-Hellman, at the same time scaring people away from 1024 bits DH: "We estimate that even in the 1024-bit case, the computations are plausible given nation-state resources". The problem of implementing DH securely are unfortunately rarely well understood. The defense approach is discussed in several RFCs [4] [5], but no paper so far take 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 an opportunist attacker a valid excuse to submit his 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. The working code along with explanations on how to reproduce our setup is included on github[6].

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 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 particular generator. 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 in section 8.

[5] Adrian et al - Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice

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

2 Attacks on Diffie-Hellman and the Discrete Logarithm

To attack a Diffie-Hellman key exchange, the attacker needs to extract the secret key a from one of the peer's public key $y_a = g^a \pmod n$. He can then compute the shared key $g^{ab} \pmod n$ using the other peer's public key $y_b = g^b \pmod n$.

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 $\frac{n}{2}$ operations to find a solution. More efficiently, algorithms that compute discrete logarithm in expected $\sqrt{q}$ steps (with q the order of the base) like Baby-step-giant-step (deterministic), Pollard rho or Pollard Kangaroo (both probabilistic) can be used. Because of the space required for Baby-step-giant-step, Pollard's algorithms are often preferred. While both are parallelizable, Kangaroo is used when the order is unknown or known to be in a small interval. For bigger groups the Index Calculus or other Number Field Sieve (NFS) algorithms are the most efficients. 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 that equation (where x is the secret key we are looking for):

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

The birthday paradox tells us that by looking for a random collision we can quickly find one in $\mathbb{O}(\sqrt{n})$. 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 iterations of the algorithm (Two different iterations of g^{xa+b} are started and end up in a loop passed a certain step) and the technique of distinghuished points is used to parallelize the algorithm (paralleled 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[7][6]: 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 $y = g^x \pmod{n}$ in subgroups of order dividing the group order. Thanks to the Chinese Remainder Theorem (CRT) stated later, the secret key can then be reassembled in the group order. Summed up below is the full Pohlig-Hellman algorithm:

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

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

2. Determine the value of x modulo $p_i^{k_i}$ for each i

3. Recompute $x \pmod{\varphi(n)}$ with the CRT

The ruse of Pohlig-Hellman's algorithm is in how do we 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 our public key to the subgroup we're looking at by computing:

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

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):

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

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 Pollard Rho to do that.

The Chinese Remainder Theorem, sometimes used for the good [8][10] will be of use here for the bad. The following theorem states why it is possible for us to find a solution to our problem once we found a solution modulo each power prime factor of the order.

[7] S. Pohlig and M. Hellman "An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance"

[8] Shinde, Fadewar Faster RSA Algorithm for Decryption Using Chinese Remainder Theorem

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 a* x *such that:*

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

And there exist a unique solution for $x \pmod{m}$

There is a simple way to recover the $x \pmod{m}$ which we will use to later reconstruct the secret key modulo the order of the group.

Proof.

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

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

□

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

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

How can we start building the value of x?

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

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

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

$$\text{If } x = a_1 m_2 \overline{m}_2 \pmod{m}$$

$$\overline{m}_2 \text{ the integer congruent to } m_2^{-1} \pmod{m_1}$$

$$\text{then } \begin{cases} x = a_1 m_2 \overline{m}_2 = \boldsymbol{a_1} \pmod{m_1} \\ x = 0 \pmod{m_2} \end{cases}$$

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:

$$\begin{array}{l} = a_2 m_1 \overline{m}_1 \pmod{m_2}) \pmod{m_2} \\ = a_2 \pmod{m_2} \end{array}$$

$$\uparrow$$

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

$$\downarrow$$

$$\begin{array}{l} = a_1 m_2 \overline{m}_2 \pmod{m_1}) \pmod{m_1} \\ = a_1 \pmod{m_1} \end{array}$$

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 has been improvement to it with the Garner's algorithm 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 send them to one peer successively to obtain his private key. It is an active attack that doesn't work against certain ephemeral protocols that renew the Diffie-Hellman public key for every new key exchange. This is for example the case of certain implementations of TLS when using ephemeral Diffie-Hellman (DHE) as a key exchange during the handshake. This variant of

Diffie-Hellman allows for *Perfect Forward Secrecy*, a mode that protects against decryption of past and future communications in case the long-term private key is stolen.
The attack is pretty straight forward and summed up below:

1. Determine the prime factorization of the order of the group
$$\varphi(n) = \prod p_i^{k_i}$$
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
$$g = \alpha^{\frac{order}{p_i^{k_i}}}$$
3. Send generators one by one as your public keys in different key exchanges
4. Determine the value of x modulo $p_i^{k_i}$ for each shared key computed
5. Recompute $x \pmod{\varphi(n)}$ with the CRT

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

3 A First Backdoor Attempt in Prime Groups

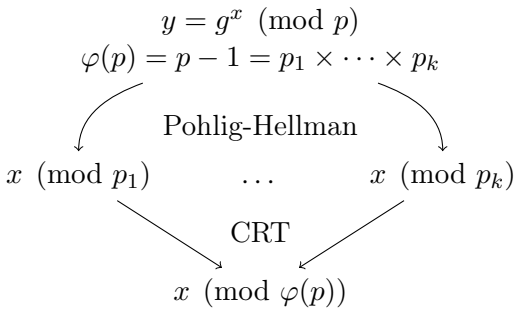
The naive approach would be to weaken the parameters enough to make the computation of discrete logarithms affordable. Making the modulus a prime of a special form ($r^e + s$ with small r and s) would facilitate the Special Number Field Sieve (SNFS) algorithm. Having a small modulus would also allow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

$$\varphi(p) = p - 1 = \boxed{p_1} \times \cdots \times p_k$$
$$\uparrow \text{order}$$
$$y = g^x \pmod{p}$$

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 the generator's "small" subgroups can be concealed modulo the

factors as well.

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.

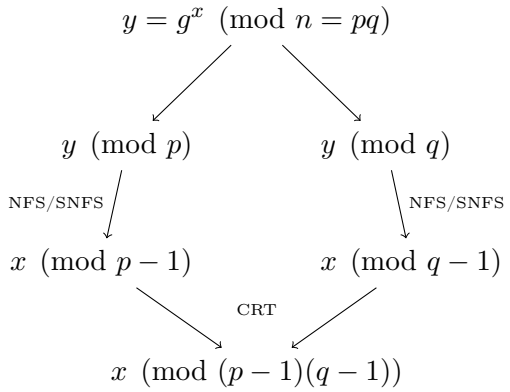


But this design is flawed in the same ways as the previous ones were: anyone can compute the order of the group $(p-1)$ and try to factor it. Anything below 300bitsis relatively easy to factor using the *Elliptic Curve Method* (ECM), a factorization algorithm which complexity only depends on the smallest factor.This lower bound on the factors make it impossible for us to use our $\mathbb{O}(\sqrt{p-1})$ algorithm which would take more than 2^{150} operations to solve the discrete logarithm of such orders.

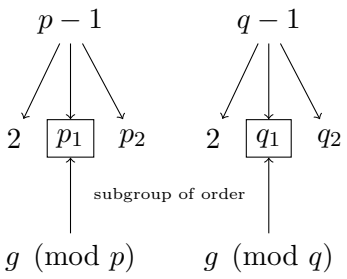
Our second contribution in section 5 uses a composite modulus to hide the smoothness of the order (CM-HSO) as long as the modulus cannot be factored. This method is preferred from the first contribution as it might only need a change of modulus. For example, in many DH parameters or implementations, $g = 2$ 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)

Let's think at our first idea in the previous section, but this time using a composite modulus $n = pq$. The discrete logarithm problem can be reduced modulo p and q and solved there before being reconstructed modulo pq with the CRT.



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 that allow us to reduce algorithms like Pollard Rho dramatically. By fixing a generator that modulo p and q generates a "small" subgroup, we would just need 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 order 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.



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 original secret key with a probability of $\frac{1}{4p_2q_2}$ which is tiny, but it doesn't matter since the shared key we will compute with that solution and the other peer's public key will be the valid shared key. This is because of the following:

Proof. Let $a + k_a p_1 q_1$ be Alice's public key for $k_a \in \mathbb{Z}$ and let $b + k_b p_1 q_1$ be Bob's public key for $k_b \in \mathbb{Z}$,
then Bob's shared key will be $(g^{a+k_a p_1 q_1})^{b+k_b p_1 q_1} = g^{ab} \pmod n$.
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. □

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.

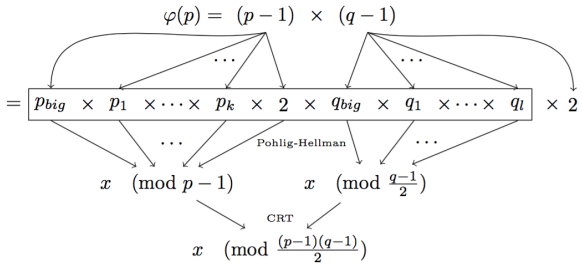
order size	expected complexity	time
40 bits	2^{20}	01s
45 bits	2^{22}	04s
50 bits	2^{25}	34s

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 a 80 bits 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 bits integer as the generator's order. For that he would need to be able to perform 2^{50} operations instantaneously if he 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 choosen large enough to resist against the same attacks as RSA. That is a n of 2048 bits with p and q both being 1024 bits long would suffise.
A problem here is that by choosing a modulus of this form, it will be close to impossible to find a fixed generator respecting these properties. 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 for example, if we want a generator $g = 2$ (which is the default hardcoded Diffie-Hellman parameter of many implementations), we would need to generate many modulus hopping that particular generator would work. The probability that it would work each time 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 $g = 2$ as a generator of our "small" subgroup. This is a problem if we want to replace secure values with our backdoored val-

ues, changing only one value (the modulus) would be more subtle than changing two values (the modulus and the generator). Our next contribution solves 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 so that the discrete logarithm is do-able 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 the union of the factors of $q-1$ and $p-1$ are pairwise co-primes and such that $p_i \leq B$ and $q_i \leq B$ for all $i \in [\![1, k]\!]$ and $i \in [[1, l]]$ 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. But one problem arises here: since $p-1$ and $q-1$ are both B-smooth, they are prone to the *Pollard's p-1 factorization* algorithm, a factorization algorithm which find a factor p if $p-1$ is partially-smooth.

To counter that, we add a big factor to each $p-1$ and $q-1$ that we will call p_{big} and q_{big} respectively.

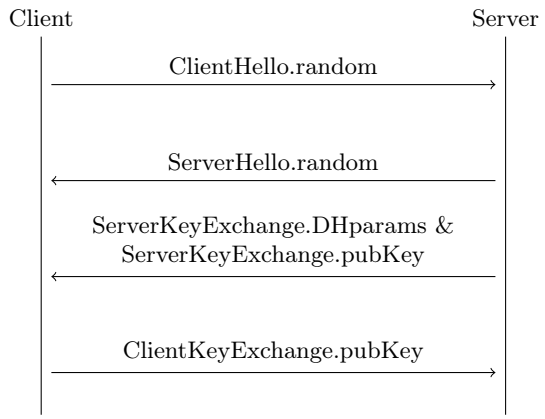
To exploit this backdoor we can reduce our public key y modulo p and q and do Pohlig-Hellman there, this is not a necessary step but this will reduce the size of the numbers in our calculations and speed up the attack. We can then recompute the private key modulo its order, which will be at a maximum $\frac{(p-1)(q-1)}{2}$.
If we look at the world's records for *Pollard's p-1* factorization algorithm[9], the maximum number used for B2 is $10^{15} \sim 50bits$ in 2015. As with our previous method, we could use a much larger factor of around 80bits to avoid any powerful adversaries and have a pleasurable 2^{40} computations on average to solve the discrete logarithm problem in these large subgroups.

6 Implementing and Exploiting the Backdoor in TLS

Theorically, any application including Diffie-Hellman could be backdoored using the previous two methods. TLS being one of the most well known example making use of Diffie-Hellman it is particularly interesting to backdoor.
Most TLS applications making use of the Diffie-Hellman algorithm for the handshake have their DH public key and parameters baked into user's or server's generated certificates. Interestingly, the parameters of the *ephemeral* version of Diffie-Hellman, used to

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

add the properties 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, which was last year the source of many discussions . This also pushed a movement of migrating to bigger parameters and increase the bitsizes of application's 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.
At the start of a new *handshake*, both the server and the client will send each other their DHE public key via a *ServerKeyExchange* and a *ClientKeyExchange* message respectively. The server will direct as well what are the DHE parameters via the same *ServerKeyExchange* message.



Let **c** and **s** be the client and the server public keys respectively. The following computation is done on each side of the exchange:

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)

To be clear, the diffie-hellman 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. The output of the PRF is sent repeatidly into another PRF with the string "key expansion" as well as 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.

6.1 Implementation

- To implement the backdoor in Socat, change a few line in file
- To implement the backdoor in OpenSSL, change a few line in file
- To implement the backdoor in Apache, change a few line in file

But to test our backdoor, it's easier to just create an asn.1 file containing our backdoor parameters and to use it as a commandline argument (examples?)

6.2 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 could be done by getting access to logs, and do a passive decryption after using the backdoor, but this was done actively in our proof of conceptby 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 forward 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 *Server Key Exchange* 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 might happen that the first messages being sent until a private key can be recovered cannot be tampered with. For better results, the work should be parallelized and the two public keys should be attacked as one might be result in a significantly faster attack. As soon as the private key of one of the 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 try sending *TLS warning alerts* that can keep a handshake alive indifinitely or for a period of time depending on the application used by both parties

7 Detecting a backdoor and protecting against one

To write in this section:

- tool testDHparams?
- statistics made on scans.io
- theory on uniform distribution
- what to do? check for safe primes, even on client side
- or only accepts a few public parameters (like ECDH)

7.1 Statistics

- We ran some tests on 50,222,805 TLS handshakes taken from scans.io from March 3rd 2016 (3n8y698qwr9ifi0e)
- 4,522,263 made use of ephemeral DH
- how many had a composite modulus?
- how many were safe primes? -> make a pie chart
- how many backdoor did we detect?

7.2 Uniform Distribution

How does a non-malicious, mistakenly, badly generated composite modulus, should be distributed (and we will later come back to this): From Handbook of Applied Cryptography fact 3.7:

Definition 1. *Let n be chosen uniformly at random form 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 $lnlnx + \mathbb{O}(1)$. (If $n = \prod p_i^{e_i}$, the total number of prime factors of n is $\sum e_i$.)*

And 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 factor of n is expected to be 7.

how to avoid backdoors:
- use only public parameters (these in RFCs), and only accept these.
- public parameter pinning (something else than public key pinning)
- if the p = 2q + 1 is not done like that (safe prime), there is a RFC that tells you how to secure such DH (safe prime, or is it Sophie Germaine prime? Or strong prime?)
- openssl dhparam (uses safe prime by default)
- also some people believe generation of prime is too difficult and that it shouldn't be possible (rfc with predefined dh groups). But then weakdh (or was it logjam rather), everybody used the same hardcoded dh prime, everybody could have/got owned
- verification of public key (but there is a patent on that? According to the diffie-hellman RFC)
- generation of safe prime

8 Conclusion and Open Problem

To write in this section:

1. backdoor in Ephemeral Elliptic Curve Diffie-Hellman Open problem
2. easy to backdoor
3. implementing DH correctly is not that hard, test for safe primes!
4. this can be implented in the client side as well, refuse non-safe primes
5. people need to verify open source more often, Socat stayed with a non-prime modulus for a year.
6. closed-source? How common is this problem?
7. people are going away from DHE and to ECDHE (https://weakdh.org/sysadmin.html)

Acknowledgements

1. stack overflow, reddit, hackernews people (check the threads to see the names)

References

[1] Bernstein, Lange and Niederhagen - Dual EC: A Standardized Back Door

[2] Checkoway et al - A Systematic Analysis of the Juniper Dual EC Incident (2016)

[3] Adrian, Bhargavan, Durumeric, Gaudry, Green, Halderman, Heninger, Springall, Thomé, Valenta, VanderSloot, Wustrow, Zanella-Béguelin, Zimmermann - Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice

[4] RFC 2631 - Diffie-Hellman Key Agreement Method

[5] RFC 2785 - Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME

[6] S. Pohlig and M. Hellman - An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance

[7] Frank Li - An Overview of Elliptic Curve Primality Proving

[8] Abelson, Sussman, Sussman - Structure and Interpretation of Computer Programs

[9] RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2

[10] Shinde, Fadewar - Faster RSA Algorithm for Decryption Using Chinese Remainder Theorem

[11] Lagrange, J. L. (1771). - Suite des réflexions sur la résolution algébrique des équations. Section troisieme. De la résolution des équations du cinquieme degré des degrés ultérieurs.

A Pohlig Hellman

One way of attacking Diffie-Hellman is to compute the discrete-logarithm x of one of the peer's public keys $y = g^x$ modulo the modulus n (here both g and n are fixed parameters). Let's take a closer look at the private exponent x. Euler's Totient Theorem states that:

Theorem 2. *If n and g are coprime positive integers, then*

$$g^{\varphi(n)} = 1 \pmod{n}$$

This tells us that the private exponent x can be written modulo $\varphi(n) = \prod p_i^{k_i}$, the order of the group.

So what if we could reduce the problem of finding $x \pmod{\varphi(n)}$ by finding $x \pmod{p_i^{k_i}}$ for each i and then easily retrieving it modulo $\varphi(n)$ with our new CRT tool? In fact a theorem says it is possible:

Theorem 3. *Let n be a composite integer. The discrete logarithm problem in $\mathbb{Z}_n^*$ polytime reduces to the combination of the integer factorization problem and the discrete logarithm problem in $\mathbb{Z}_p^*$ for each prime factor p of n.*

Note that in theory we can't reduce our public key to an element of a subgroup if we're already in a smaller group (we cannot "escape" or subgroup) or if we're in a subgroup that doesn't include the whole subgroup we're trying to get into. We cannot "escape" a group, we can only go deeper in a smaller subgroup already included in our group.