Report

Implementing the Logjam Attack: Exploiting Export Ciphers in TLS

Kryptanalytische Angriffe auf Internetprotokolle

by

Burkhardt, Hadrian
Derwisch, Oliver
Goßen, Daniel
Lockenvitz, Niko

Supervisor

Prof. Dr. Christian Dörr
Cybersecurity - Enterprise Security

Hasso Plattner Institute at University of Potsdam

August 13, 2021

Contents

1 Introduction **1**

2 The Logjam Attack **2**
- 2.1 High-level Idea . . . 2
- 2.2 TLS Handshake and Attack Steps . . . 2
- 2.3 Feasibility of Discrete Log Computation . . . 4
- 2.4 Implications of the Logjam Attack . . . 4

3 Implementation of the Logjam Attack **5**
- 3.1 General Architecture . . . 5
- 3.2 TLS Parsing and Manipulation in MITM . . . 5
- 3.3 Computing the Discrete Log . . . 6
- 3.4 Demonstration of Two Attack Types . . . 7
- 3.5 Firefox Demo . . . 8

4 Implementation Challenges **10**
- 4.1 TLS Parsing and Manipulation . . . 10
- 4.2 Challenges For a Browser Demonstration . . . 10
- 4.3 Challenges for Discrete Log Computation . . . 11
  - 4.3.1 Cluster . . . 11
  - 4.3.2 CADO-NFS . . . 13

References **15**

1 Introduction

In 2015, Adrian et al. [1] reported a vulnerability that was affecting TLS and therefore basically all major web browsers. Their Logjam attack exploits an already known vulnerability in the Diffie-Hellman key exchange in regard to the computation of discrete logs and the fact that a considerable amount of web servers supported export-grade DH groups.

In this report, we will first explain how this attack works in chapter 2. After that, we outline the main aspects of our implementation of this attack in chapter 3. There, we not only describe the implementation of the attack itself, but also how it can be used to exploit web applications and how this can be presented. Finally, in chapter 4 we cover major challenges we faced during the implementation and how we solved them.

2 The Logjam Attack

In this chapter, we will discuss how the Logjam attack [1] works, list the needed requirements for a successful attack, and show the consequences Logjam causes. For in-depth details of the attack, we additionally recommend the original paper by Adrian et al. [1].

2.1 High-level Idea

The key idea is to break the Diffie-Hellman Ephemeral (DHE) key exchange by computing the discrete log of the server's public key, i.e. the ephemeral secret key. The only way this is feasible is when the DHE prime is small enough. To force servers to use such small DHE primes, an man-in-the-middle (MITM) attacker manipulates the TLS handshake to downgrade the TLS session so that export ciphers are used and correspondingly only a 512 bit prime is chosen.

Therefore, the primary requirement to realize Logjam is a server that supports an old cipher suite with a small prime ($\leq$ 512 bit) for the DHE key exchange. Ideally, the server uses a standardized and widely used DHE prime, as the main workload of the attack only depends on the used prime. Thus, if multiple servers use the same prime, all those could be attacked with almost the same amount of computation power.

2.2 TLS Handshake and Attack Steps

If those requirements are met, the attack works as explained in the following section. To support our explanations, Figure 2.1 shows a regular DHE TLS handshake and Figure 2.2 the difference if a Logjam MITM attacker is present.

The MITM starts to manipulate the `ClientHello` and explicitly requests only an export cipher that the server offers. In this way, the server will choose a small prime for the DHE cipher suite. However, towards the client, the MITM manipulates the `ServerHello` such that the client thinks one of the DHE cipher suites offered is confirmed.

Next, the server shares its certificate, group parameters for the DHE calculation, and its public key g^{Y_S}. This traffic is only forwarded by the MITM.

At this stage, the client cannot detect that the server uses a different cipher suite, as both used cipher suites perform DHE and there is no visible difference. In addition, no integrity check of the negotiated cipher suites happened so far. The client simply sees a server that chose a small DHE group for one of the proposed and seemingly accepted cipher suites. The client therefore accepts the group parameters and continues.

While the client now provides its part to the DHE key exchange, the MITM can already start computing the secret key that corresponds to the server's public key. After the key exchange, the client indicates that the communication from now on should be protected

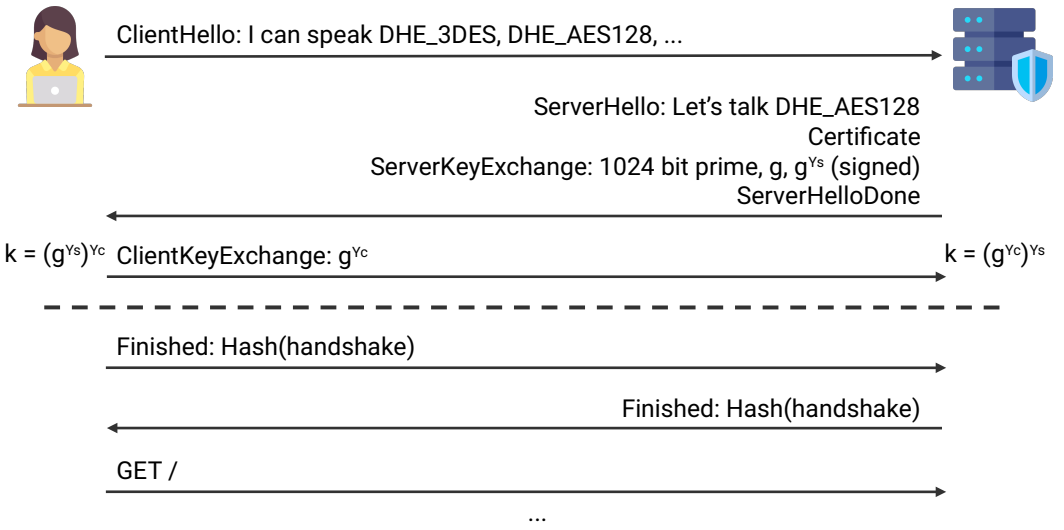

Figure 2.1: Regular DHE TLS handshake (icons by Freepik from www.flaticon.com).

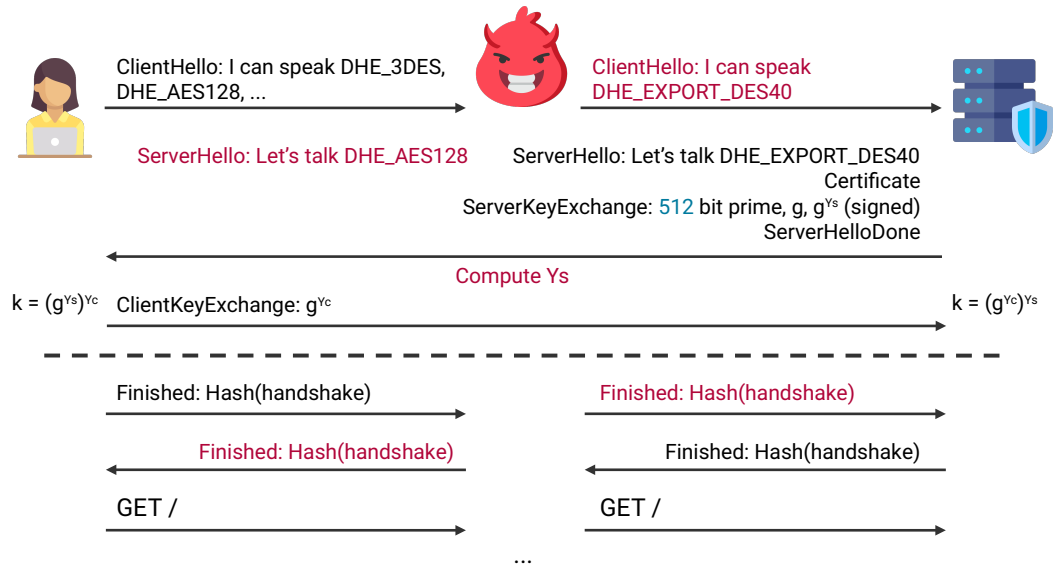

Figure 2.2: DHE TLS handshake with manipulation by MITM (icons by Freepik from www.flaticon.com).

using the negotiated cipher suite by sending the `ChangeCipherSpec` message. Therefore, the following `Finished` message is encrypted and integrity-protected. It basically includes a hash of the handshake to make sure that client and server saw the same handshake and now MITM manipulated the handshake.

Now, the MITM needs to use the computed server's secret key to reconstruct the TLS session's master secret and forge this `Finished` message. Even though it might sound conflicting, it does not contradict the purpose of the `Finished` message, as this can only protect against attackers that do not know the session's master secret. But as the computation of the server's secret key happens within seconds, the MITM can forge arbitrary messages and freely manipulate the communication between client and server.

Similarly, the server's `ChangeCipherSpec` is forwarded and the `Finished` message is manipulated by the MITM. The following messages, e.g. application data, are re-encrypted as different cipher suites are used between client and MITM and MITM and server.

2.3 Feasibility of Discrete Log Computation

The main reasons why the attack is possible are the small DHE group size of just 512 bit and the re-use of such groups. This is because most parts of computing the discrete log only depend on the prime and can therefore be pre-computed (see Figure 2.3). The remaining computation can be done in real time during the handshake.

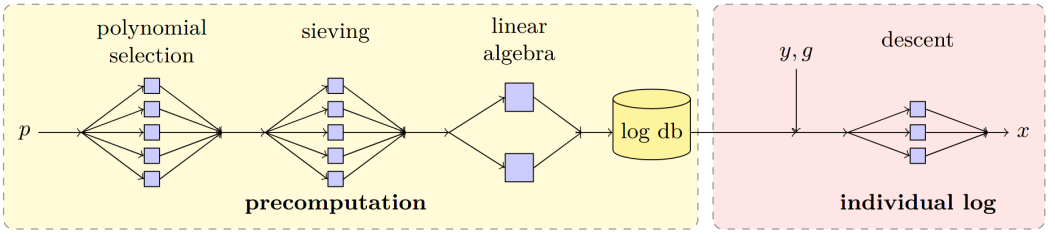

Figure 2.3: Four steps of computing the discrete log using the number field sieve algorithm [1]

Furthermore, according to Adrian et al., back in 2015, around 80 % of the servers vulnerable to this attack used the same 512 bit prime. Because of that, with a single pre-computation, one was not only able to attack a single server but roughly 70,000 of Alexa Top 1M HTTPS domains [1].

2.4 Implications of the Logjam Attack

The Logjam attack shows that breaking DHE key exchange can be significantly optimized by pre-computation if DHE primes are re-used. For small primes, where the individual log computation can be performed within seconds, even active attacks and arbitrary manipulation of the encrypted communication's content are possible. For larger primes, only passive attacks are possible, still allowing to decrypt the communication afterwards. Responding to the attack, major browser vendors increased the minimum size of DHE primes to 1024 bit, hoping that this makes active attacks infeasible.

3 Implementation of the Logjam Attack

3.1 General Architecture

To implement the Logjam attack, we first set up a network of Docker containers, where each major component runs in an own container. This includes among others the server, which is configured to support secure cipher suites as well as export ciphers with small DHE primes, the client, and the MITM. Packets between client and server are automatically redirected via the MITM as this is configured directly inside these containers. Then, the MITM parses and manipulates the TLS connection. For that, it communicates with cado-nfs which is used to compute the discrete log. This can either be run on a local computer in just another Docker container, where computing the discrete log for 128 bit primes is feasible in a few seconds, or on a cluster, where this can be done for 512 bit primes. Further details on the discrete log computation are presented in section 3.3.

The code for all components can be found in our Git repository[1]

3.2 TLS Parsing and Manipulation in MITM

All packets between client and server are sent via the MITM. To be able to parse and manipulate them, they are provided to a Python script by a corresponding *iptables* rule using *nfqueue*. The Python script reads the content of TCP packets using *scapy* and handles them appropriately.

For that, the Python script implements all relevant TLS components for the attack on a byte-level so that we can extract all required information and do the necessary manipulations. This includes reading `client_random` and the suggested cipher suites from the `ClientHello`, setting these cipher suites to only `TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA` for the `ClientHello` that is sent to the server, reading `server_random` and the selected cipher suite from the `ServerHello`, altering the chosen cipher suite to a DHE cipher suite that was proposed by the client, forwarding the server's `Certificate`, reading the server's DHE public key from the `ServerKeyExchange` as well as forwarding the `ServerHelloDone`, the `ClientKeyExchange`, and the client's `ChangeCipherSpec`.

As the communication is now encrypted, the result of cado-nfs is used to compute the TLS master secret, from which all cryptographic keys can be derived. To forge the `Finished` message, the client's and the server's handshake view (i.e. including our manipulations) is maintained and then used to compute `verify_data` using a hash function and a PRF corresponding to the used TLS version. When manipulating the client's `Finished` message, the server's handshake view is confirmed. After the server's `ChangeCipherSpec` is forwarded, the same is done for its `Finished` message, where now the client's view of the handshake gets confirmed.

[1] See `https://github.com/nikolockenvitz/logjam`

The application data that is sent in the following is decrypted, printed, potentially manipulated, encrypted again, and then forwarded. How this is used to exploit a web application is presented in section 3.4.

To be able to manipulate multiple TLS sessions at the same time, all state variables are stored in a context object that is loaded based on the connection information, e.g. client IP and port.

3.3 Computing the Discrete Log

As computing the discrete log and especially the required pre-computation demands huge resources, we worked on the HPI Future SOC Lab cluster, which provided us with about 600 CPUs. The cluster consists of 15 Nodes, each with 40 CPU cores, and 1 TiB RAM. Slurm is the workload manager, and there is also one slurmsubmit-node used to submit Slurm jobs, distributing the workload across the cluster.

To do the pre-computations for the discrete logarithm problem, we chose to use cado-nfs. This tool uses the number field sieve to factorize big numbers.

The idea of a number-field-sieve is

> based on Fermat's factoring method:
>
> - Find two integers x and y such that $x^2 \equiv y^2$ (mod N)
> - With good probability, $gcd(x \pm y, N)$ gives a non-trivial factor of N
>
> Obtain such equalities through two number fields
>
> - f_1 and $f_2 \in \mathbb{Z}[X]$ two polynomials, irreducible and co-prime over $\mathbb{Q}$
> - α_i root of $f_i : \mathbb{Q}(\alpha_i)$ is an algebraic number field
> - f_1 and f_2 chosen such that they have a common root m in $\mathbb{Z}/N\mathbb{Z}$

[2, p. 8]

The whole process of solving the discrete logarithm problem consists of four stages. The first three stages pre-compute a database of correlated logarithms for a specific prime p. The last stage derives the individual logarithm for a target in real-time.

Polynomial selection In this stage cado-nfs tries to find suitable functions for f_1 and f_2 from that cado-nfs can derive the factor bases. In our case, we used the default cado-nfs script, which offers good parallelization for multiple nodes in this stage. The runtime for polynomial selection was approx. 7,600 core-hours (approx. 2 days)

Sieving This step collects relations, purges unnecessary ones, and finally, constructs a compressed relation matrix. For this stage, we also used the cado-nfs script, which also offered good parallelization for multiple nodes. The runtime for sieving was approx. 21,400 core-hours (approx. 5 days)

Linear algebra A database of discrete logarithms of a specific density is constructed from the relation matrix to recover as many logarithms as possible. The resulting database for 512 bit is about 2.5 GB in size. We did not use the cado-nfs default script for this step, since it uses only one node for the computation. Instead, we tried to run this stage with a customized bash file and MPI support, which could distribute the workload among the cluster nodes; however, the runtime increased drastically, so we were better off using a single node. Unfortunately, the default cado-nfs script could not pick up work from the checkpoints of the customized script. So, we had to use customized scripts further to finish the following stages, see 4.3.2. The runtime for this step was approx. 15,000 core hours (approx. 2 weeks).

Descent approx The last step is a lookup of correlated logarithms in the database to do the final computation of the individual log of a given target; this stage is not part of the pre-computation and is supposed to run in real-time. Here we also used a customized script to derive the individual log(target) and noticed that the lookups are not efficient (see 4.3.2) so that computing the result took longer than expected.

3.4 Demonstration of Two Attack Types

The Logjam attack can now be used to read and manipulate, and thus control, the data traffic between server and client. To actually exploits this, a simple login page and a counter were implemented for demonstration purposes. We used an old version of Firefox (version 35) to access these pages in the web browser. As soon as the web page is requested in the web browser, the MITM starts the attack, computes the server's secret key, and intercepts and manipulates the session at will. For both examples, the further steps of the MITM are quite different.

Eavesdropping In our first attack, we are bugging the data traffic between client and server. When the login page is called up, a GET request from the client and the subsequent transmission of the website from the server are decrypted on the MITM on the fly, making the content visible. That is, based on the HTML code, it is easy to see that it is a login page and our victim wants to log in to a service. Next, the victim logs in. So, the victim sends a POST request with username and password to the server. Upon successful login, the MITM now knows the victim's credentials. Since the victim can now successfully execute a GET request for its profile, the MITM also sees the victim's decrypted and presumably sensitive data. As the credentials are leaked, the MITM can arbitrarily use the account and even try the credentials for other servers where the victim might have reused the password.

JavaScript Injection In the second attack, it is demonstrated how malicious code can be injected into a website. The client requests a simple web page that displays a counter. As the counter only works with JavaScript, we can be sure that this in enabled in the client's browser. The MITM injects further JavaScript code that continuously connects to a Command & Control (C&C) server, which is controlled by the attacker. This C&C server is just a Flask server that waits for user input and forwards it to the client's web browser.

Thus, it is a simple remote shell that allows executing arbitrary code in the client's browser. For demonstration purposes, we inject code to compute Fibonacci numbers client-side and also alter the displayed HTML. Furthermore, an alert message is displayed, warning the client that the computer is allegedly infected with malware and asking the client to call the Linux customer support.

The former attack is well suited to steal sensitive data like credentials or credit card information that is sent via this presumable secure connection. The latter attack is perhaps even more critical. In our demo, we "only" stole minor computing resources and caused an alert dialogue to pop up. However, such attacks are often the gateway for the placement of malware or the permanent use of computing power, such as for mining cryptocurrencies.

3.5 Firefox Demo

As explained in the previous section, we used an actual web browser for the demonstration to show the potential implications of the Logjam attack. In contrast to TLS traffic via terminal tools such as `curl`, one visits interactive HTML pages that may require filling forms or can contain extended client-side functionality such as JavaScript. That is why, the web browser is required for it and we can not rely on terminal-based tools.

As all current major web browsers at least require DHE cipher suites with 1024 bits, the setup has to use old browsers. All other components are included in our Docker images, but web browsers require a GUI so a separate virtual machine (VM) needs to be set up.

For local demonstrations with limited computing power, the Rekonq browser of Kubuntu 13.10 is suitable, as this browser also works with very small group sizes such as 128 bit. However, the actual attack of Adrian et al. [1] attacks cipher suites of 512 bit. With a powerful external computer for the discrete log computation step, this allows a demonstration with Firefox 35.0 that requires larger group sizes. Rekonq is a niche browser and the general setup is identical to that for Firefox, so Firefox is used in the following paragraphs as example browser.

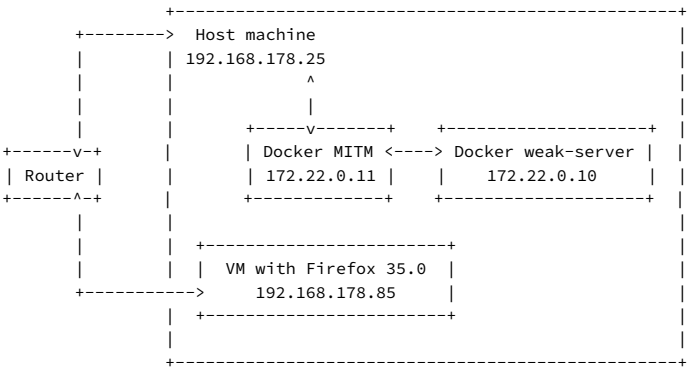

Figure 3.1: Network architecture for the Firefox demo.

Figure 3.1 shows the involved components and routing configuration. Using Linux network routing rules, the traffic of Firefox inside the VM is routed via the host machine's

operating system and via the MITM to the weak server. The technical details of the setup and all necessary old software can be found in our Git repository. It also contains a video[2] with a demonstration of:

- The MITM steps for each TLS handshake message (breaking the discrete log with pre-computation for a 512 bit group takes approximately one minute);
- The MITM capability to read all TLS encrypted traffic, such as credentials, in plaintext;
- Execution of client-side code controlled via the attacker's command and control server.

[2] See resources/tls-export-ciphers-logjam-demo-512-bit-firefox.mp4 in the repository

4 Implementation Challenges

4.1 TLS Parsing and Manipulation

One major problem in the beginning was finding the required tools to establish weak connections between client and server and to manipulate this communication purposefully. To be able to still support 512 bit primes, we chose Ubuntu 14.04 as a base image for the three main Docker containers. However, this also limited the Python version and the packages that could be used. Even with old versions, tools like curl or openssl still demanded at least 512 bit primes which was challenging for testing as only for 128 bit primes the discrete log computation was feasible on a desktop computer. So we also needed to modify the client's openssl to support any size for the DHE primes. For the parsing and manipulation of the TLS protocol, we finally decided to do this without any library (except for the cryptographic operations) since we found no tool that met our needs. For example, scapy also supports parsing TLS, however this does not directly work in the Python version we are using and additionally the documentation of it is rather sparse.

One aspect where scapy works rather good is the manipulation of TCP/IP packet headers, i.e. checksums and lengths as well as sequence and acknowledgment numbers. But as packets are sometimes exceeding the Ethernet MTU, just replacing the incoming packet's content with the new content is not always possible. Hence, for larger packets, manual fragmentation is required. In such cases, only the last bytes of a packet are sent via manipulating the original one, accepting it and sending it via nfqueue. The previous blocks are sent before that using a simple socket.

A challenge related to the own implementation of the TLS parsing from scratch is that the relevant details are scattered over several RFCs. For example, while TLS 1.2 is used by default, TLS 1.1 includes details on the relevant cipher suites and TLS 1.0 includes details on the specifics of export ciphers. Furthermore, Firefox 35, which we used for the live demo, adds a TLS Next Protocol Negotiation Extension which lead to the MITM failing to correctly parse and manipulate the handshake. To fix this with least effort, we decided to just remove this extension from the `ClientHello` so that the server does not respond to it and Firefox no longer sends corresponding messages.

4.2 Challenges For a Browser Demonstration

First, getting old browsers that were released five to ten years ago running is already a challenge. Modern web browsers use several libraries, e.g. for cryptographic operations. These need to be installed in the corresponding old software version. Compiling all necessary components of a browser is thus very frustrating and complicated. Fortunately, Firefox offers an FTP server with an archive of all released Firefox versions in binary form,

packaged together with all necessary software dependencies. For our local demonstration with less than 512 bit cipher suites, we managed to download an old Kubuntu 13.10 image from archive.org. This image already ships with the needed Rekonq browser that accepts small DHE group sizes of e.g. 128 bit.

Next, this old web browsers cannot be integrated into our existing Docker setup as Docker cannot provide a full GUI. Therefore, a separate VirtualBox VM is necessary. The network traffic between weak server and old web browser needs specific routing to pass our MITM. This is explained in section 3.5 and Figure 3.1. For easier routing rules, the VM operates in bridged network mode and thus receives a separate IP address by the router. Under certain network conditions, such as in the HPI Wi-Fi, the Wi-Fi clients cannot communicate with each other in the internal network. To compensate for this problem during the live demonstration, we used an Android smartphone and its hotspot functionality to create a Wi-Fi without such routing problems.

While we managed to demonstrate the attack locally for 128 bit groups, the demo for 512 bit requires significantly more computational power. The final step of cado-nfs needs a separate, powerful server to finish fast enough before timeouts of the server and/or web browser trigger. We called one node of the FSOC cluster to perform this operation.

As even the FSOC cluster node needs between 60 and 80 seconds for the final step of the computation, the server timed out during the TLS handshake. We took the easiest option and increased the server timeout. In the wild, it would of course not be possible for the attacker to manipulate server-side settings. However, using clever techniques such as sending TCP keepalives or deliberately fragmenting packets, it would also be possible to extend the available time for the computation. Adrian et al. [1] discussed some of them in their paper.

Furthermore, a small modification in the client's Firefox was required as part of the demonstration. The communication with the command and control server happens, for simplicity, without TLS but just over a normal HTTP connection. As this results in a mixed content error, we simply disabled this error in `about:config` so that the browser no longer prevents loading simple HTTP traffic when the actual connection was established using HTTPS. Of course, in a realistic attack, this communication would also be secured with TLS, and additionally, the IP of the command and control server would not be hard-coded but set dynamically and use some kind of proxy in between.

4.3 Challenges for Discrete Log Computation

4.3.1 Cluster

Compiling Binaries Since we were working on an external cluster, we could not install software as we wanted. For that reason, cado-nfs and it's dependencies had to be compiled locally on the slurmsubmit-node, where the individual cluster nodes could then access the binaries via shared folders. For distributing the workload among the cluster nodes, the cluster uses Slurm as a workload manager, and some libraries need additional compiler flags to work.

- **hwloc-2.4.1** no additional compiler flags.

- **openmpi** use the version already installed on the system, since the version needs to be compatible with already installed software on the cluster.
- **gmp-6.2.1** no additional compiler flags.
- **ecm-7.0.4** –with-gmp=<gmpdir>, linking the locally built gmp library.
- **cado-nfs-3.0.0** edit path to dependencies, and settings for large numbers in local.sh.

It is also important to check whether all libraries that are required are also available on the cluster nodes. The ldd command can help you to ensure that:

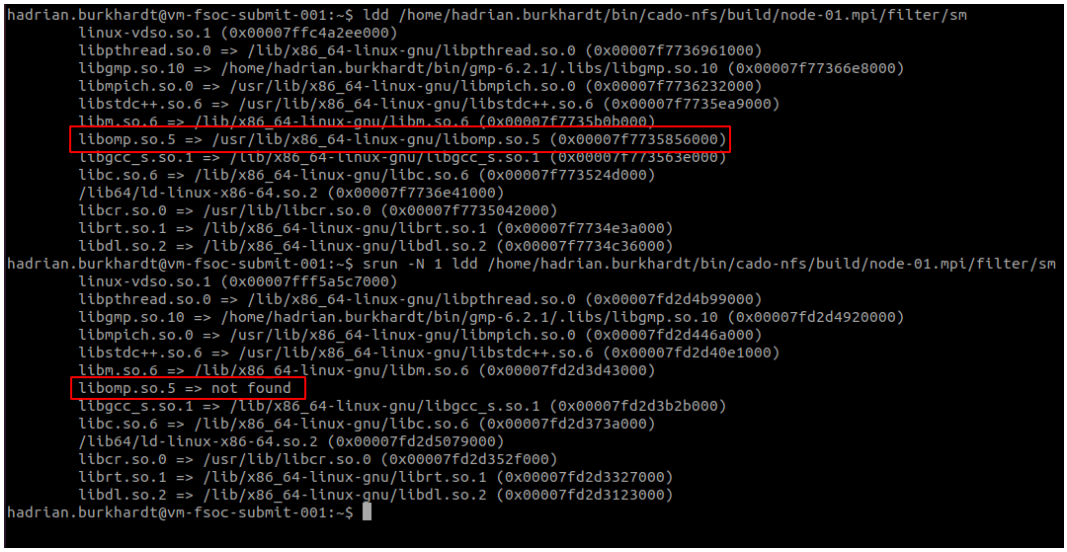

Figure 4.1: Missing omp library on a cluster node.

Using Slurm Slurm is a workload manager that can be tricky to use. Make sure that one reads the documentation or man-pages of the installed version. Furthermore, monitor the workload distribution and confirm that the observed behavior is intentional. To allocate a certain node with specified resources, use the *-B 4:10:2* (four sockets, ten cores per socket, two threads per core). To make sure that the program can use the reserved resources, use *-c 80* (80 threads per task). The full command to launch a program on 5 nodes, 1 task per node, with MPI support is: *srun –mpi=openmpi -N 5 -B 4:10:2 -n 5 -c 80 <prog>*

Disk Space It is crucial for large-scale computation to have enough disk space that is accessible by all nodes to store intermediate results persistent. We also recommend that one estimates required hardware resources and runtime of such projects whenever possible. Comparing estimations against observations will help to identify unwanted behavior early on.

Connectivity Issues It is possible that the remote cluster fails or is unavailable for several reasons. Therefore, it is crucial to save one's progress in checkpoint files, where one can resume work after dealing with the issue.

Using MPI MPI is a Message Passing Interface that enables high performance on supercomputers similar to a node cluster. However, in our case, using MPI did not give a reasonable improvement in runtime. The letdown in runtime could be related to the cluster setup or cado-nfs which does not provide efficient tasks for MPI-binaries. Indeed, in our case, running code on multiple nodes via MPI is slower than running the code without MPI on a single node. It is recommended not to use MPI at all or to develop an MPI capable design if the processing power of multiple nodes is needed, such as in the pre-computation for 768 bit DH primes.

4.3.2 CADO-NFS

The aggregated scripts of cado-nfs provide a solid base for medium-sized number factorization out of the box. It is possible to distribute workload to several nodes during the early stages of polynomial selection and sieving. However, the linear algebra stage runs on a single node by default. Due to our initial estimations, doing the linear algebra on one only node with 40 cores, would take about 60 days; hence, we tried to split the workload across the cluster with MPI. As described above, this approach failed but luckily, our initial estimate did not yield true. So, that running the linear algebra on a single node was sufficient for our experiments to finish in 2 weeks. However, on a larger scale, one would want to efficiently split the linear algebra workload across multiple nodes.

Writing your customized scripts If one works on a larger scale, we recommend writing one's customized version of execution scripts. Combining the original cado-nfs scripts and customized execution commands could result in corrupt database files and should be avoided. Moreover, one must choose the parameters carefully; we used the default `params.p155` file, and we could only recover 97.5 % of the logarithms. Consequently, as multiple of these logarithms are required to compute the target's logarithm, only 30 % of the attacks were successful. For script and parameter developing, helpful information is provided in one of cado-nfs' repositories[3]

Converting Results The security of the DHE key exchange relies on the fact that it is hard to find an integer x that satisfies the following equation for a given prime p and a generator g, which is a primitive root modulo p.

$$g^x \equiv a \ (mod\ p)$$

Unfortunately, cado-nfs does not give the solution to the equation above, but computes the discrete logarithm of a given target modulo ell, where ell is the largest factor of the group order $p-1$. For the groups generated by openssl, this gives $ell = (p-1)/2$.

$$log_r(target) \equiv x \ (mod\ ell)$$

$$log_r(target) \equiv x \ (mod\ \frac{p-1}{2})$$

[3] https://gitlab.inria.fr/cado-nfs/records/-/tree/master/

The generator base r of this result is arbitrary, so next we need to rescale the result to our generator base, which is $g = 2$ for openssl's groups. To do that we can use the following equation to change the generator base:

$$log_g(target) = \frac{log_r(target)}{log_r(g)}$$

$$log2: \ log_r(2) \equiv x \ (mod\, ell)$$

$$result: \ log_2(target) \equiv log_r(target) \cdot invmod(log2, ell) \ (mod\, ell)$$

At this point we are finished if our result satisfies $2^{result} \equiv target \ (mod\ p)$. If not, we need to proceed further and compute the final result candidates using the Chinese remainder theorem ($CRT(a_1, \ n_1, a_2, n_2)$). This follows the Pohlig-Hellman algorithm, where the intermediate results, i.e. the logarithms in regards to the factors of the group order $p - 1$, are combined using the CRT. As the remaining factor of $p - 1$ is 2, we can solve this by exhaustive search, that means trying both possible values 0 and 1.

$$result_0 : CRT(result, ell, 0, 2) \ mod \ (p - 1)$$

$$result_1 : CRT(result, ell, 1, 2) \ mod \ (p - 1)$$

Either $result_0$ or $result_1$ satisfies the equation $2^{result} \equiv target \ (mod\ p)$ and the discrete logarithm problem is solved correctly, otherwise it failed.

Real-time Computation and Timeout The last challenge we had to face was that our attack should work in real-time, which means that we could actively manipulate traffic of a TLS session instead of only decrypting and reading its contents later on, especially as this active manipulation is required to downgrade the DHE prime's size to something that is feasible to attack. Since the final computation in our attack still takes a while, we changed the server's timeout (default: 60 s) to 5 minutes. However, the computation of the individual logarithm in cado-nfs suffers from poor implementation and lacks efficient parallelization. With some small optimizations we were able to decrease the time for this final computation step from 150 s to 80 s on average. One can argue that increasing the server's timeout would not be necessary when additionally using a mature database that supports multi-threaded and concurrent lookup methods. From the MITM point of view, there is also the possibility of delaying the transmission of client packets to the server, buying more time to look up the pre-computed logarithms.

Bibliography

[1] 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: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pages 5–17. DOI: 10.1145/2810103.2813707.

[2] Jérémie Detrey. *Factoring integers with CADO-NFS*. URL: http://www.ens-lyon.fr/LIP/AriC/wp-content/uploads/2015/03/JDetrey-tutorial.pdf (visited on Aug. 13, 2021).