15:12 Nail in the Java Key Store Coffin

by Tobias "Floyd" Ospelt

The Java Key Store (JKS) is Java's way of storing one or several cryptographic private and public keys for asymmetric cryptography in a file. While there are various key store formats, Java and Android still default to the JKS file format. JKS is one of the file formats for Java key stores, but the same acronym is confusingly also used the general key store API. This article explains the security mechanisms of the JKS file format and how the password protection of the private key can be cracked. Due to the unusual design of JKS, we can ignore the key store password and crack the private key password directly.

By exploiting a weakness of the Password Based Encryption scheme for the private key in JKS, passwords can be cracked very efficiently. As no public tool was available exploiting this weakness, we implemented this technique in Hashcat to amplify the efficiency of the algorithm with higher cracking speeds on GPUs.

The JKS File Format

Examples and API documentation for developers use the JKS file format heavily, without any security warnings.[44] This format has been the default key store since key stores were introduced to Java. As early as 1999, JDK 1.2 introduced the "-much stronger" JCEKS format that uses 3DES.[45] However, JKS remained the default format. Just to mention some examples, Oracle databases and the Apache Tomcat webserver still use the JKS format to store their private keys.

When building an Android 7 app in the Android Studio IDE, it will create a JKS file with which to self-sign the app. Every application on Android needs to be signed before it can be installed on a device, and the phone will check that an update for an app is signed with the same key again. The private keys generated by Android Studio are valid for 25 years by default. Android does not offer any recovery mechanism to recover a lost private key, so efficient cracking of JKS files also benefits developers who forgot their passwords.

The JKS format is due to be replaced by PKCS12 as the default key store format in the upcoming Java 9.[46] When talking to members of the security community who can still remember the nineties, some seem to remember that JKS uses some kind of weak cryptography, but nobody remembers exactly. Let's explore weaknesses of the JKS file format and what an attacker needs to extract a private key in cleartext.

When a new key store is created and a new keypair generated, the developer has to set at least two passwords. There is not only a password for the key store as a whole (key store password), but each private key in it has its own password as well (private key password), while public keys do not have passwords. Both passwords are used independently. Surprisingly, the key store password is not used to encrypt any parts of the JKS file format, it is only used for integrity protection. This means the encrypted private key bytes and the cleartext bytes of public keys in a key store can be extracted without knowing the key store password.[47] The password of the private key however, is used to apply a custom Password Based Encryption to the private key. Having two passwords leads to three possible cases.

In the first case, there is a password on the key store, but no private key password is used. (In practice, the available Java APIs prevent this.) However, in such a key store the private key would not be protected at all.

The second case is when the key store password and the private key password are identical. This is very common in practice and the default behavior of most tools such as Java's `keytool` command. If no separate password for the private key is specified, the private key password will be set to the key store password.

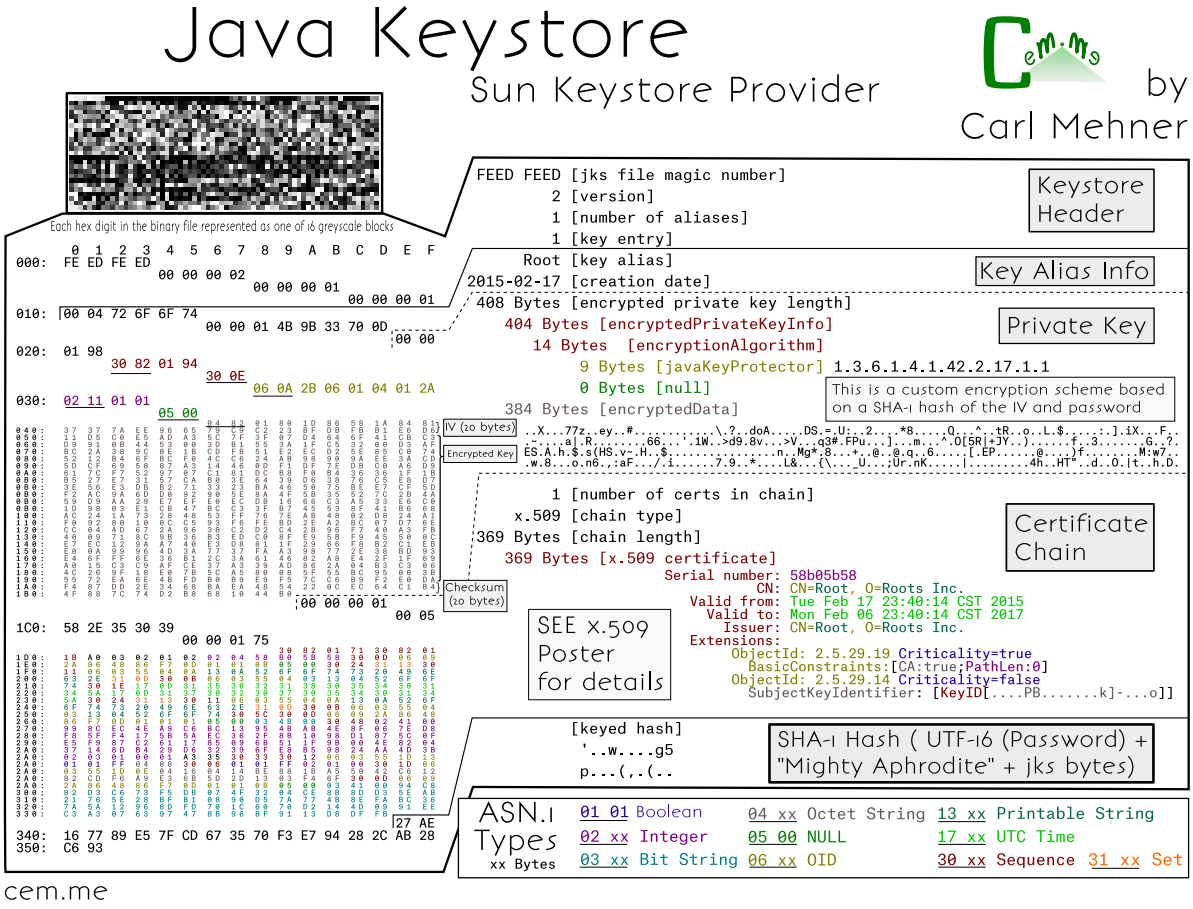
In the third case, both passwords are set but the

[44] `http://docs.oracle.com/javase/6/docs/api/java/security/KeyStore.html#getDefaultType()`
`http://download.java.net/java/jdk9/docs/api/java/security/KeyStore.html#getDefaultType--`
`https://developer.android.com/reference/java/security/KeyStore.html#getDefaultType()`
`http://stackoverflow.com/questions/11536848/keystore-type-which-one-to-use`
`http://www.pixelstech.net/article/1408345768-Different-types-of-keystore-in-Java----Overview`

[45] See Dan Boneh's notes on JCE 1.2 from CS255, Winter of 2000.

[46] `http://openjdk.java.net/jeps/229`

[47] `https://gist.github.com/zach-klippenstein/4631307`



key store password is not the same as the private key password. While not the default behavior, it is still very common that users choose a different password for the private key.

It is important to demonstrate that in the third case some password crackers will crack a password that is useless and cannot be used to access the private key. The Jumbo version of the John the Ripper password cracking tool does this, cracking the (useless) key store password rather than the private key password. Let's generate a key store with different key store (`storepass`) and private key password (`keypass`), then crack it with John:

```
$ keytool -genkey -dname                                    \
  'CN=test, OU=test, O=test, L=test, S=test, C=CH'          \
  -noprompt -alias mytestkey -keysize 512                   \
  -keyalg RSA -keystore rsa_512.jks                         \
  -storepass 1234567 -keypass 7654321
$ pypy keystore2john.py rsa_512.jks > keystore.txt
$ /opt/john-1.8.0-jumbo-1/run/john                          \
  --wordlist=wordlist.txt keystore.txt
[...]
1234567          (rsa_512.jks)
[...]
```

While this reveals the `storepass`, we cannot access the private key with this password. My proof of concept will crack the private key password instead:[48]

```
$ java -jar JksPrivkPrepare.jar rsa_512.jks > privkey.txt
$ pypy jksprivk_crack.py privkey.txt
Password: '7654321'
```

Naive Password Cracking

If we take the perspective of an attacker, we can conclude that we will not need to crack any password in the first case to get access to the private key. In theory, it also doesn't matter which password we find out in the second case, as both are the same. And in the third case we can simply ignore the key store password; we only need to crack attack the private key password.

However, when we encounter the second case in practice, we would like to use the most efficient

[48] `unzip -j pocorgtfo15.pdf jksprivk/JksPrivkPrepare.jar jksprivk/jksprivk_crack.py`

password cracking technique to find the key store password or the private key password. This means we need to explore first how each password can be cracked individually and which one leads to the most efficient cracking method.

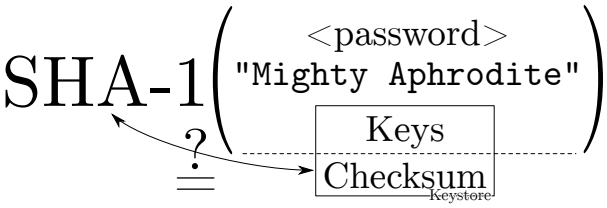
There are already several programs that will try to crack the password of the key store:

- John the Ripper (JtR) Jumbo version[49] extracts necessary information with a Python script and the cracking is implemented in C;
- KeyStoreBrute[50] tries to load the key store via the official Java method in Java;
- KeystoreCracker[51] uses the simple official Java way in Java as well;
- keystoreBrute[52] uses `keytool` on the command line with the `storepass` option (subprocess);
- bruteforcer.py[53] uses `keytool` on the command line with the `storepass` option (subprocess);
- Patator[54] uses `keytool` on the command line with the `storepass` option (subprocess).

All these parse the JKS file format first, which has a SHA-1 checksum at the end. They then calculate a SHA-1 hash consisting of the password, the magic "`Mighty␣Aphrodite`" and all bytes of the key store file except for the checksum. If the newly calculated hash matches the checksum, it was the correct password.

No other operation with the key store password takes place when parsing the JKS file format; therefore, we can conclude that this password is only used for integrity protection. When the correct password is guessed and it is the same as the private key password, an attacker can now decrypt the private key.

From a performance perspective, this means that for every potential password a SHA-1 hash needs to be calculated of nearly all bytes of the key store file. As key stores usually hold private and public keys of at least 512-byte length, the SHA-1 hash is calculated over several thousand bytes of input. To summarize, the effort to check one password for validity is roughly:



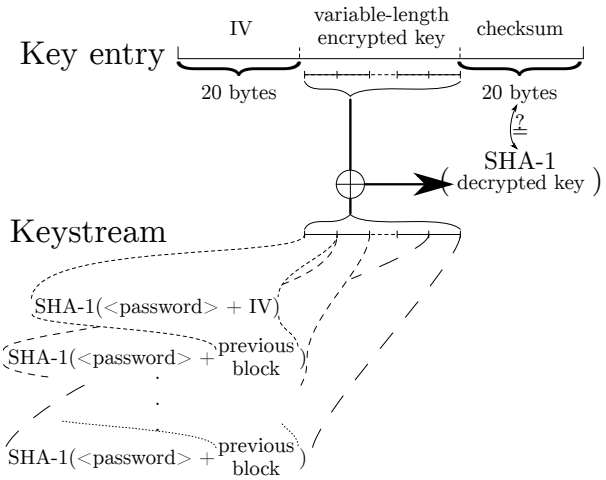
It is also important to emphasize again that the above implementations will waste CPU time if the key store password is not identical to the private key password (third case) and are not attempting to crack the password necessary to extract the private key.

There are also implementations that crack the password of the private key directly:

- android-keystore-recovery[55] tries to decrypt the entire private key with each password, in Scala;
- android-keystore-password-recover[56] tries to decrypt the entire private key with each password, in Java.

These implementations have in common that they parse the JKS file format, but then only extract the entry of the encrypted private keys. For each private key entry, the first 20 bytes serve as an Initialization Vector and the last 20 bytes are again a checksum. The implementations then calculate a keystream. The keystream starts as the SHA-1 hash of the password plus IV. For every 20 bytes of the encrypted private key, the next 20 bytes of the keystream are calculated as the SHA-1 of the password plus previous keystream block (of 20 bytes). The encrypted private key bytes are then XORed with the keystream to get the private key in cleartext. This is a custom Password Based Encryption (PBE) scheme with chaining. As a last step, the cleartext private key is SHA-1 hashed again and compared to the checksum that was extracted from the JKS private key entry. Therefore, the effort to check one password for validity is roughly:

[49] `http://www.openwall.com/lists/john-users/2015/06/07/3`
[50] `git clone https://github.com/bes/KeystoreBrute`
[51] `git clone https://github.com/jeffers102/KeystoreCracker`
[52] `git clone https://github.com/volure/keystoreBrute`
[53] `https://gist.github.com/robinp/2143870`
[54] `https://www.darknet.org.uk/2015/06/patator-multi-threaded-service-url-brute-forcing-tool/`
[55] `https://github.com/rsertelon/android-keystore-recovery`
[56] `https://github.com/MaxCamillo/android-keystore-password-recover`



Efficient Password Cracking

From a naive perspective, it was not analyzed which of these algorithms would be more efficient for password cracking.[57] However, an article on Cryptosense.com was published in 2016[58] and didn't seem to get the attention it deserves. It points out that for the private key password cracking method it is not necessary to calculate the entire keystream to reject an invalid password. As the cleartext private key will be a DER encoded file format, the first SHA-1 calculation of password plus IV with the XOR operation is sufficient to check if a password candidate could potentially lead to a valid DER encoded private key. These all miss out on this optimization and therefore do too many SHA-1 calculations for every password candidate.

It turns out, it is even possible to pre-calculate the XOR operation. For each password candidate only one SHA-1 hash needs to be calculated, then some bytes of the result have to be compared to the pre-calculated bytes. If the bytes are identical, this proves that the password might decrypt the key to a DER format. Practical tests showed that a DER encoded RSA private key in cleartext will start with `0x30` and bytes at index six to nineteen will be `0x00300d06092a864886f70d010101`. Similar fingerprints exist for DSA and EC keys. These bytes we expect in a DER encoded private key can be XORed with the corresponding encrypted private key bytes to precalculate the SHA-1 output bytes we are looking for.

This means, the cracking can be optimized to use a more efficient two-step cracking algorithm to crack the private key password. After parsing the JKS file format and precalculating the necessary values, we have the following optimized algorithm:

0. Choose a password in pseudo UTF-16, meaning that a null byte is added to every character.
1. `keystream = SHA-1(password + STATIC_20_BYTES_IV_FROM_PRIVKEY_ENTRY)`
2. Check if bytes at index 0 and 6 to 19 of the keystream correspond to `PRECOMPUTED_15_BYTES_DER_PROOF`. If they are not the same, go to step 0.
3. Let `keybytes` be every 20 bytes of `STATIC_VARIABLE_LEN_ENCRYPTED_BYTES_FROM_PRIVKEY_ENTRY`.
4. For each `keybytes`:
   (a) `key += keystream` $\oplus$ `keybytes`
   (b) `keystream = SHA-1(password‖keystream)`
5. `checksum = SHA-1(password‖key)`
6. Check if `checksum` is `STATIC_20_BYTES_CHECKSUM_FROM_PRIVKEY_ENTRY`. If they are the same, key is the private key in cleartext and we can stop. Otherwise, go to step 0.

As practical tests will later indicate, step 3 is typically never reached with an incorrect password during cracking and all passwords can be rejected early. In fact, Hashcat only implements steps 0 to 3, as the probability that a wrong candidate is ever found is neglectible ($1/2^{120}$)!

Implementation

The parsing of the file format and extraction of the precomputed values for cracking were implemented as a standalone JAR Java version 8 command line application `JksPrivkPrepare.jar`. The script will

[57] While the key store calculations must do the single SHA-1 over all bytes of the public and private keys in the key store, the private key calculations are many more SHA-1 calculations but with less bytes as inputs.

[58] Might Aphrodite – Dark Secrets of the Java Keystore

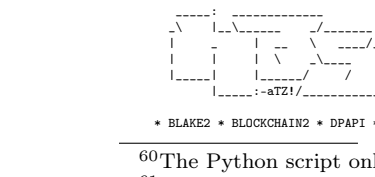
[59] Running much faster with the PyPy Python implementation rather than CPython. The script works without further dependencies. However, another script in the benchmark section needs the numpy packet. It has to be installed for PyPy. The easiest way of installing is usually via PIP: `pypy -m pip install numpy`

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_123456.jks           \
  -storepass 123456 -keypass 123456
$ java -jar JksPrivkPrepare.jar rsa_512_123456.jks > privkey_123456.txt
$ pypy -m cProfile -s tottime jksprivk_naive_crack.py privkey_123456.txt
Password: '123456'
         10278681 function calls (10277734 primitive calls) in 9.763 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   123457    2.944    0.000    2.944    0.000 jksprivk_naive_crack.py:14(xor)
  2345683    1.651    0.000    1.651    0.000 {method 'digest' of 'HASH' objects}
  2345684    1.608    0.000    1.608    0.000 {_hashlib.openssl_sha1}
  2345683    1.491    0.000    5.266    0.000 jksprivk_naive_crack.py:19(get_keystream)
[...]
$ pypy -m cProfile -s tottime jksprivk_crack.py privkey_123456.txt
Password: '123456'
         649118 function calls (648171 primitive calls) in 0.438 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   123476    0.086    0.000    0.086    0.000 {method 'digest' of 'HASH' objects}
   123477    0.067    0.000    0.067    0.000 {_hashlib.openssl_sha1}
        1    0.056    0.056    0.293    0.293 jksprivk_crack.py:54(get_candidates)
       14    0.055    0.004    0.486    0.035 __init__.py:1(<module>)
[...]
```

Figure 11. Java Key Store with a Short Password

prepare the precomputed values for a given JKS file and outputs it as asterix separated values.

As a PoC, a Python script `jksprivk_crack.py`[59] was implemented to do the actual cracking of the private key password. To put a final nail in the coffin of the JKS format, it is important to enable the security community to do efficient password cracking.[60] To optimize cracking speed, Jens "atom" Steube — developer of the Hashcat password recovery program — implemented the cracking step in GPU optimized code. Hashcat takes the same arguments as the Python cracking script. As hashcat uses a weakness in SHA-1,[61] the cracking speed on a single NVidia GTX 1080 GPU reaches around 7.8 (stock clock) to 8.5 (overclocked) billion password tries per second.[62] This allows to try all alphanumeric passwords (uppercase, lowercase, numbers) of length eight in about eight hours on a single GPU.



Benchmarking

When doing a benchmark, it is important to try to measure the actual algorithm and not some inefficiency of the implementation. Some simple measurements were done by implementing the described techniques in Python. All the mentioned resources are available in the feelies.[63] Let's first look at the naive implementation of the private key cracker `jksprivk_naive_crack.py` versus the efficient private key cracking algorithm `jksprivk_crack.py`. Let's generate a test JKS file first. We can generate a small 512-byte RSA key pair with the password 123456, then crack it with both implementations. Both implementations only try numeric passwords, starting with length 6 password 000000 and incrementing, as in Figure 11.

These measurements show that a lot more calls to the update and digest function of SHA-1 are necessary to crack the password in the naive script. If the keysize of the private key in the JKS store is bigger, the time difference is even greater. Therefore, we conclude that our efficient cracking method is far

[60] The Python script only reaches around 220,000 password-tries per second when run with PyPy on a single 3-GHz CPU.
[61] `https://hashcat.net/events/p12/js-sha1exp_169.pdf`
[62] `git clone https://github.com/hashcat/hashcat`
[63] `unzip -j pocorgtfo15.pdf jksprivk/jksprivk_resources.zip`

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_12345678.jks          \
  -storepass 12345678 -keypass 12345678
$ java -jar JksPrivkPrepare.jar rsa_512_12345678.jks > privkey_12345678.txt
$ pypy -m cProfile -s tottime jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
         116760228 function calls (116759281 primitive calls) in 60.009 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 23345699   16.940    0.000   16.940    0.000 {_hashlib.openssl_sha1}
 23345698   16.082    0.000   16.082    0.000 {method 'digest' of 'HASH' objects}
 23345775   10.971    0.000   10.972    0.000 {method 'join' of 'str' objects}
        1    8.560    8.560   59.851   59.851 jksprivk_crack.py:54(get_candidates)
 23345698    4.024    0.000    4.024    0.000 {method 'update' of 'HASH' objects}
 23345679    3.274    0.000   14.245    0.000 jksprivk_crack.py:91(next_brute_force_token)
 [...]
$ pypy /opt/john-1.8.0-jumbo-1/run/keystore2john.py rsa_512_12345678.jks \
  > keystore_12345678.txt
$ pypy -m cProfile -s tottime jkskeystore_crack.py keystore_12345678.txt
Password: '12345678'
         163420866 function calls in 84.719 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 70037037   33.712    0.000   33.712    0.000 {method 'update' of 'HASH' objects}
 23345679   17.780    0.000   17.780    0.000 {method 'digest' of 'HASH' objects}
 23345680   12.022    0.000   12.022    0.000 {_hashlib.openssl_sha1}
 23345682    9.679    0.000    9.679    0.000 {method 'join' of 'str' objects}
        1    8.482    8.482   84.716   84.716 jkskeystore_crack.py:14(crack_password)
 23345679    3.042    0.000   12.721    0.000 jkskeystore_crack.py:26(next_brute_force_token)
 [...]
```

Figure 12. Java Key Store with a Longer Password

more suitable.

Now we still have to compare the efficient cracking of the private key password with the cracking of the key store password. The algorithm for key store password cracking was also implemented in Python: `jkskeystore_crack.py`. It takes a password file as argument like John the Ripper does. As these implementations are more efficient, let's generate a new JKS with a longer password, as shown in Figure 12.

In this profile, we see that the update method of the SHA-1 object when cracking the key store takes much longer to return and is called more often, as more data goes into the SHA-1 calculation. Again, the efficient cracking algorithm for the private key is faster and the difference is even bigger for bigger key sizes.

So far we tried to compare techniques in Python. As they use the same SHA-1 implementation, the benchmarking was kind of fair. Let's compare two vastly different implementations, the efficient algorithm `jksprivk_crack.py` to John the Ripper. First, create a wordlist for John with the same numeric passwords as the Python script will try, then run the comparison shown in Figure 13.

That figure shows that John is faster for 512-bit keys, but as soon as we grow to 1024-bit keys in Figure 14, we see that our humble little Python script wins the race against John. It's faster, even without John's fancy C code or optimizations!

As John the Ripper needs to do SHA-1 operations for the entire key store content, the Python script outperforms John the Ripper. For larger key sizes, the difference is even bigger.

These benchmarks were all done with CPU calculations and Hashcat will use performance optimized GPU code and Markov Chains for password generation. Cracking a JKS with private key password `POC||GTFO` on a single overclocked NVidia GTX 1080 GPU is illustrated on Figure 15.

Neighborly Greetings

Neighborly greetings go out to atom, vollkorn, cem, doegox, ange, xonox and rexploit for supporting this article in one form or another.

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_12345678.jks          \
  -storepass 12345678 -keypass 12345678
$ java -jar JksPrivkPrepare.jar rsa_512_12345678.jks > privkey_12345678.txt
$ time pypy jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
        54.96 real        53.76 user         0.71 sys
$ pypy /opt/john-1.8.0-jumbo-1/run/keystore2john.py rsa_512_12345678.jks           \
  > keystore_12345678.txt
$ time /opt/john-1.8.0-jumbo-1/run/john --wordlist=wordlist.txt keystore_12345678.txt
[...]
12345678         (rsa_512_12345678.jks)
[...]
        42.28 real        41.55 user         0.33 sys
```

Figure 13. John the Ripper is faster for 512-byte keystores.

```
$ time pypy jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
        58.17 real        56.36 user         0.84 sys
$ time /opt/john-1.8.0-jumbo-1/run/john --wordlist=wordlist.txt keystore_12345678.txt
[...]
12345678         (rsa_1024_12345678.jks)
[...]
        64.60 real        62.96 user         0.57 sys
```

Figure 14. For 1024-bit keystores, our script is faster (full output in the feelies).

```
$ ./hashcat -m 15500 -a 3 -1 '?u|' -w 3 hash.txt ?1?1?1?1?1?1?1?1?1
hashcat (v3.6.0) starting...
[...]
* Device #1: GeForce GTX 1080, 2026/8107 MB allocatable, 20MCU
[...]
$jksprivk$*D1BC102EF5FE5F1A7ED6A63431767DD4E1569670...8*test:POC||GTFO
[...]
Speed.Dev.#1.....:  7946.6 MH/s (39.48ms)
[...]
Started: Tue May 30 17:41:56 2017
Stopped: Tue May 30 17:50:24 2017
```

Figure 15. Cracking session on a NVidia GTX 1080 GPU.