CSci 4271W (011 and 012 Sections) Lab Instructions

Lab 14 **May 5th, 2024**

Ground Rules. You may choose to complete this lab in a group of up to three students. Before you leave the lab, **make sure you have submitted to Gradescope, you included all group members on the submission, and the autograder found all required files!**

1 A Hands-on Replay Attack

In a previous lab, we showed an attack on a network protocol (Mal-in-the-middle relaying) in the form of a packaged tool. In today's lab, instead, you'll implement a very simple attack (a replay) from yourself against a simple authentication protocol. To make things easier to debug, we've implemented the protocol in a text-based format that can be passed through Unix pipes. We'll pass the protocol data over the network in TCP streams using netcat (`nc`), a useful tool for using TCP connections like pipes.

The programs this week implement an authentication protocol based on challenge-response and public-key signatures. The purpose of the protocol is for a client to authenticate to a server. The server and the client both have secret signing keys for a public-key signature scheme, and the corresponding verification keys are also known to the other party.

On the most abstract level, the protocol looks like this:

```
S -> C: (Chall, Sign_S(Chall))
C -> S: (Chall, Sign_S(Chall), Sign_C(Chall), T, N)
```

The server sends the client a "challenge" which is a nonce, together with a signature to prove that the challenge came from the server. Then the client signs the same challenge with its own signing key, and returns it together with the original challenge and server signature, a timestamp, and a fresh nonce. The sever verifies both its own signature on the challenge (proving that it's a challenge the server sent) and the client's signature (proving that the client produced the signature). It also checks that the timestamp T is recent and that the client-chosen nonce N has never been used before. The timestamp and the client-chosen nonce prevent basic replay attacks. However, this protocol turns out to still be vulnerable to a slightly more complicated kind of relay attack. Can you see what it is?

We've implemented the steps of this protocol in three separate programs named `gen-challenge`, `do-auth`, and `check-auth`. `gen-challenge` and `check-auth` together simulate the server. The program `gen-challenge` produces the first message, while `check-auth` implements the server's checking of the second message. `do-auth` implements the client's behavior: it takes the output of `gen-challenge` and produces the corresponding response. The challenges and nonces are 256 bits, the public key signatures use the elliptic curve algorithm Ed25519, and the timestamps are in the Unix format of the number of seconds since the start of 1970. The challenges, nonces, and signatures are base-64 encoded, while the timestamp is in decimal. Each part of a message is on a separate line. `check-auth` keeps its list of previously-used nonces in a file named `.4271_replaylab_nonce_db` in your home directory (if you'd like you can delete it when you're done with the lab).

To make this available as a network service, two shell scripts named `gen-challenge-server.sh` and `check-auth-server.sh` wrap the corresponding parts of the server side processing as TCP

services that listen on ports 4271 and 4272 respectively. In other words, a client requests a challenge by connecting to port 4271, and then sends a response by connecting to port 4272. For initial testing we'll recommend you run everything on your VM, but if your attack is working you can also try it out against a server we're running on VM 101. To simulate the fact that you as a network attacker would not have access to the client and servers' signing keys, we have pre-compiled the programs so you can run them but not see they keys inside.

1.1 Authentication service in action

You can download the necessary files for this lab using:

```
$ git clone https://github.umn.edu/badly-coded-alpha/replaylab.git
$ cd replaylab
```

Both servers need to be running at the same time as the client, so you can run them in separate terminals are just in the background, as in:

```
$ ./gen-challenge-server.sh &
[1] 500727
$ ./check-auth-server.sh &
[2] 500730
```

To check that the challenge generator is running correctly, you can use the command `nc localhost 4271`, which opens a TCP connection to port 4271 on the current machine connected to its standard input and standard output. We don't need to supply any input to get a challenge, so you can redirect the input from the empty device `/dev/null`. You should see something like this:

```
$ nc localhost 4271 </dev/null
FEkDCrgklvKhBsKhQ8p1YEv2ViXivSjYmDMFu3yPWfk
fDbv/UPMFpa+d5K8PPYe4oGtYihRwuDj3XDl0Kh+KGTJajn3MTEc2vqqhffPURegVUT11CDoO9s2R2Vm5KdQAQ
```

(Of you didn't include `</dev/null`, you can interrupt netcat after seeing the output using Ctrl-C or Ctrl-D.) Except, of course, you will see two different random-looking strings. You'll also see different strings (a fresh challenge) if you run the command again.

To show the protocol working with the legitimate client, we can run the `do-auth` program with its input from port 4271 and its output to port 4272; the response on port 4272 will tell us if the authentication succeeded. That looks like:

```
$ nc localhost 4271 </dev/null | ./do-auth | nc localhost 4272
Signature verification succeeded for challenge
Signature verification succeeded for response
Timestamp check succeeded
Nonce uniqueness check succeeded
Authentication succeeded!
```

However, the authentication will fail if the timestamp is more than 5 seconds out of date, which we can illustrate by waiting 6 seconds to send the response to server:

```
$ nc localhost 4271 </dev/null | ./do-auth | (sleep 6; nc localhost 4272)
Signature verification succeeded for challenge
Signature verification succeeded for response
Timestamp check failed: stale
```

Also, the server will reject a response that has a nonce it has seen before:

```
$ nc localhost 4271 </dev/null | ./do-auth >saved.resp
$ cat saved.resp| netcat localhost 4272
...
Authentication succeeded!
$ cat saved.resp| netcat localhost 4272
...
Duplicate nonce (DB line 11)
```

(You have to run all three commands within 5 seconds to see this behavior. You can do this if you're used to using the up-arrow command history in the shell, but you can also put all the commands on one line separated by semicolons.)

1.2 Building a replay attack

Your goal for the lab is to implement a more sophisticated replay attack that works. Based on observing a successful authentication, as in the first two steps in the example above, transform the previous response into a new response that will be accepted as fresh. You may find it helps to think about what parts of the response you can generate yourself as the attacker, and which ones only the legitimate server or client can create.

Continuing with the Unix pipe theme of the lab, a convenient way to implement your attack is a program that takes an out-of-date, already-used response (like `saved.resp` in the example above) as input, and produces the fresh replay-attack response. To get you started, we've written a Python script named `make-reply.py` that attempts to do this. You can run it like this:

```
$ cat saved.resp | python make-replay.py | nc localhost 4272
Old response is 453 seconds out of date
Signature verification failed for challenge
```

However, as you see, the version of this script that we've provided doesn't implement a working attack. You'll need to change what message it creates to get a working attack.

1.3 Extra information, bonus attack

If you're curious to see the implementations of the steps of the protocol, we've put copies of their original C code in the lab directory as `gen-challenge-nokey.c`, `do-auth-nokey.c`, and `check-auth.c`. Note however that the programs with `nokey` in the names have had the secret key information redacted. The programs are compiled with the LibSodium cryptography library with `-lsodium` on the compiler command line.

We've also left a copy of the port 4271 and 4272 servers running another one of the VMs: the one with NNN=101, whose full hostname is `csel-xsme-s25-csci4271-101`. If your attack works against your own copies of the servers, it should also work against this one, just putting the

hostname in place of `localhost` in the `nc` commands. Note however that this server uses different signing keys, and we haven't given you a corresponding `do-auth` binary. The only resource you'll have to work with is the file `saved-101.resp`, which is a single saved response for this server. You won't be graded on whether you can do this version of the attack.

1.4 Cleaning up

If you ran the servers in separate terminals, you can kill them with Ctrl-C when you are done. If you ran them in the background, you can use the `jobs` and `kill` commands to kill them, as in:

```
$ jobs
[1]-  Running                 ./gen-challenge-server.sh &
[2]+  Running                 ./check-auth-server.sh &
$ kill %1
$
[1]-  Terminated              ./gen-challenge-server.sh
$ kill %2
$
[2]+  Terminated              ./check-auth-server.sh
```

The `%1` and `%2` correspond to the `[1]` and `[2]` in the output of `jobs`: they'd only be other numbers if you had run other background programs in between. The blank command lines indicate that sometimes you have to wait a moment before the `Terminated` messages are printed, since `kill` doesn't wait for its victim to die.

2 All done!

The only thing you'll need to submit is your working version of the `make-replay.py` script. Copy it from your VM to your CSE Labs home directory in the usual way, then submit it to the lab assignment on Gradescope. Make sure you include all of the members of your group!

Once you've submitted your file, the autograder will test to make sure the proper file was submitted, and notify you if anything went wrong, within a few minutes. (We'll manually check your submissions after the deadline.)

Congratulations, you've finished Lab 14, the final lab for CSci 4271W!