Shawyoun Shaidani Tufts University

Comp 116, Final Project: Supporting Material for “Attacks and Defenses against VoIP”

“Zorg” is an open-source ZRTP implementation sponsored by PrivateWave, an Italian firm which uses Zorg as part of its voice encryption solution. It is written in Java, and it can be deployed on Android, Blackberry, or a J2SE server. Below I have included some of the most important code fragments, in order to demonstrate how the key exchange negotiation works on a lower level. The entire code base may be downloaded at https://github.com/opentelecoms-org/zrtp-java.

In what is known as the Discovery phase, the communication begins with one party sending a “Hello” message containing their ZID (a unique identifier for the party), the version of ZRTP they are running, and which algorithms they support:

```
private byte[] createHelloMsg() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    int len = 3;
    baos.write(createMessageBase(MSG_TYPE_HELLO, len));
    baos.write(VERSION.getBytes());
    String clientID = new String(CLIENT_ID_RFC); // Must be 16 bytes
                                                 // long
    baos.write(clientID.getBytes("US-ASCII"));
    baos.write(hashChain.H3);
    byte[] zid = localZID;
                            ...

    baos.write(zid);
    String hashes = "";
    byte hc = 0;
    // Support for SHA-384 and SHA-256
    if (TestSettings.KEY_TYPE_EC38) {
        hashes += "S384";
        ++hc;
    }
    if (TestSettings.KEY_TYPE_EC25 || TestSettings.KEY_TYPE_DH3K) {
        hashes += "S256";
        ++hc;
    }
    String ciphers = DEFAULT_CIPHERS; // AES-128 & 256
    String authTags = authMode.name(); // HMAC-SHA1 32

                            ...

    String keyTypes = "";
    byte kc = 0;
    if (TestSettings.KEY_TYPE_EC38) {
        keyTypes += "EC38";
        ++kc;
    }
    if (TestSettings.KEY_TYPE_EC25) {
        keyTypes += "EC25";
        ++kc;
    }

    if (TestSettings.KEY_TYPE_DH3K) {
        keyTypes += "DH3k";
        ++kc;
```

Commented [s1]: “boas” will hold the Hello string we’re trying to build; we add to it as components are calculated

Commented [s2]: Represents vendor and release of the ZRTP software

Commented [s3]: 256-bit hash of a nonce unique to each party (used to check for false packets used in DDoS attacks)

Commented [s4]: This is a random 96-bit ID generated when ZRTP is installed; it’s unique to every device and used to identify endpoints in a session

Commented [s5]: List of hash algorithms supported by this local ZRTP instance; we check our settings to see if any should be added to the list. These, along with similar lists later in this method, are ultimately used to negotiate what algorithms both parties will use (by looking at the intersection of the lists & picking the fastest one)

Commented [s6]: All ZRTP endpoints must support DH3K, so this is always true. DH3K means that the encryption is based on the prime factorization of numbers that are integer multiples of a 3072-bit number. The higher the number of bits, the more secure the algorithm.

Commented [s7]: List of ciphers supported by this instance of ZRTP. Advanced Encryption Standard-128 & 256 are required for all instances.

Commented [s8]: List of key agreement types supported by this instance of ZRTP. All implementations must support DH3K.

```
        }
        int cipherCount = ciphers.length() / 4;
        int authModeCount = authTags.length() / 4;
        String sasTypes = new String("B256"); // 32 bit & 256 bit sas supported
        int sasTypeCount = sasTypes.length() / 4;
        // Signature type field will be length 0 as no signatures used
        baos.write(0x00); // SMP flags all set to zero
        baos.write(hc); // hc = hashes count
        baos.write((cipherCount << 4) | authModeCount); // cc = cypher count = 2
        baos.write((kc << 4) | sasTypeCount); // kc = key agreement, sc = SAS count = 1
        baos.write(hashes.getBytes());
        baos.write(ciphers.getBytes());
        baos.write(authTags.getBytes());
        baos.write(keyTypes.getBytes());
        baos.write(sasTypes.getBytes());
        byte[] hello = baos.toByteArray();
        baos.close();
```

Commented [s9]: The Short Authentication String (SAS) is presented to the UI to the two users, who can verbally compare it during their session to detect MiTM attacks. This list contains the types of SAS that are supported for this instance of ZRTP.

Commented [s10]: Add all the “supported algorithm” lists to the buffer

Commented [s11]: Pass the buffer to an array “hello” which serves as a middle man for the final output

...

```
        byte[] msg = addImplicitHMAC(hello, hashChain.H2);
        if (TestSettings.TEST && TestSettings.TEST_ZRTP_WRONG_HMAC_HELLO)
{
            randomGenerator.getBytes(msg, hello.length, 2);
        }
        if (platform.isVerboseLogging()) {
            logBuffer("HELLO MSG:", msg);
        }
        return msg;
    }
```

Commented [s12]: Msg contains the “hello” array from before, appended with a SHA-256-HMAC hash of that entire message (minus the hash itself). This helps protect against false “hello” messages by correlating this media stream with a specific “hello” message. The total resulting string is the final output.

When the Hello message is received by the other party, they store the information within it, and then respond with a HelloACK message. This message does not contain any results of the negotiations, although negotiation is already taking place internally. It simply exists to prevent further Hello messages from being sent:

```
    private void doHello(byte[] data, int offset, int len) throws IOException {
```

...

```
        if (rxHelloMsg != null) {
            if (rxHelloMsg.length != len
                    || !platform.getUtils().equals(rxHelloMsg, 0, data, offset,
                            len)) {
                raiseDenialOfServiceWarning("Hello message differs from the accepted Hello");
                return;
            }
            // Already seen the Hello message, just ACK it & move on
            sendHelloACK();
```

Commented [s13]: Under normal conditions, Hello messages should be the same from the same party. Otherwise, we might be experiencing a DDoS attack. Do not respond to the message and raise a flag instead.

Commented [s14]: We’ve seen a repeat Hello message, but it’s legitimate. This can happen if the other party hasn’t received an acknowledgment for some reason. Try to send a HelloACK again. (The sendHelloACK method does little besides create a packet with the string “HelloACK” within it.)

```
        } else {
            byte[] aMsg = extractData(data, offset, len);

                                    ...
            farEndZID = extractData(aMsg, 64, 12);
            int hashCount = aMsg[77] & 0x0F;
            int cipherCount = (aMsg[78] >>> 4) & 0x0F;
            int authCount = aMsg[78] & 0x0F;
            int keyCount = (aMsg[79] >>> 4) & 0x0F;
            int sasCount = aMsg[79] & 0x0F;
            int hashPos = 80;
            int cipherPos = hashPos + (hashCount * 4);
            int authPos = cipherPos + (cipherCount * 4);
            int keyPos = authPos + (authCount * 4);
            int sasPos = keyPos + (keyCount * 4);
            boolean isLegacyAttributeList = false;
            hashMode = HashType.SHA256;

            for (int i = 0; i < hashCount; i++) {
                // Only need to check for SHA-384 as, if its not there, we'll
                // always use SHA-256
                if (DH_MODE_EC_USE_256 && TestSettings.KEY_TYPE_EC25) {
                    if (platform.getUtils().equals(HashType.SHA256.getType(),
                                0, aMsg, hashPos + i * 4, 4)) {
                        hashMode = HashType.SHA256;
                    }
                } else if (!DH_MODE_EC_USE_256 && TestSettings.KEY_TYPE_EC38) {
                    if (platform.getUtils().equals(HashType.SHA384.getType(),
                                0, aMsg, hashPos + i * 4, 4)) {
                        hashMode = HashType.SHA384;
                    }
                }
                if (platform.isVerboseLogging()) {
                    logString("HELLO MSG - HASH: "
                            + new String(aMsg, hashPos + i * 4, 4));
                }
            }

            isLegacyAttributeList = LegacyClientUtils.checkHash(platform,aMsg, hashPos, hashCount);

            // If cipherCount == 0, only supports mandatory AES-128
            cipherInUse = CipherType.AES1;
            for (int i = 0; i < cipherCount; i++) {
                // Only need to check for AES3 as, if its not there, we'll
                // always use AES1
                if (platform.getUtils().equals(CipherType.AES3.getSymbol(), 0,
                        aMsg, cipherPos + i * 4, 4)) {
                    cipherInUse = CipherType.AES3;
                }
                if (platform.isVerboseLogging()) {
```

Commented [s15]: Now begins the taking of data from the Hello message itself. We start by parsing the Hello message to establish the locations of each relevant field.

Commented [s16]: With the locations of each field established, we begin iterating through them all to find out what algorithms are supported by the other party. We compare them to the ones supported by us, and set the fields to the intersectioning algorithms as appropriate. (This is the so-called "negotiation"). The first algorithm that is negotiated is the hash algorithm.

Commented [s17]: These are Booleans representing our own settings.

Commented [s18]: Here we look at the other party's hash algorithms. Based on this logic, we set our hashMode to a certain type (SHA256 by default).

Commented [s19]: The second algorithm we are negotiating is the Cipher Type, which is AES1 by default.

Commented [s20]: If the other party supports AES3, use that instead.

```
                        logString("HELLO MSG - CIPHER: "
                                + new String(aMsg, cipherPos + i * 4, 4));
                    }

                    authMode = AuthenticationMode.HS32;
                    for(int j = 0; j < authCount ; j++) {
if(platform.getUtils().equals(AuthenticationMode.HS80.getSymbol(), 0, aMsg,
authPos + j*4, 4)) {
                            authMode = AuthenticationMode.HS80;
                        }
                        if(platform.isVerboseLogging()) {
                            logString("HELLO MSG - AUTH MODE: " + new
String(aMsg, authPos + j*4, 4));
                        }
                    }
                }

                isLegacyAttributeList &=
LegacyClientUtils.checkCipher(platform ,aMsg, cipherPos, cipherCount);

                // If keyCount == 0, only supports mandatory DH3K
                dhMode = KeyAgreementType.DH3K;
                for (int i = 0; i < keyCount; i++) {
                    if (DH_MODE_EC_USE_256 && TestSettings.KEY_TYPE_EC25) {
                        if (platform.getUtils().equals(
                                KeyAgreementType.ECDH256.getType(), 0, aMsg,
                                keyPos + i * 4, 4)) {
                            dhMode = KeyAgreementType.ECDH256;
                        }
                    } else if (!DH_MODE_EC_USE_256 &&
TestSettings.KEY_TYPE_EC38) {
                        if (platform.getUtils().equals(
                                KeyAgreementType.ECDH384.getType(), 0, aMsg,
                                keyPos + i * 4, 4)) {
                            dhMode = KeyAgreementType.ECDH384;
                        }
                    }
                    if (platform.isVerboseLogging()) {
                        logString("HELLO MSG - KEY: "
                                + new String(aMsg, keyPos + i * 4, 4));
                    }
                }
                dhSuite.setAlgorithm(dhMode);

                isLegacyAttributeList &=
LegacyClientUtils.checkKeyAgreement(platform ,aMsg, keyPos, keyCount);

                // If sasCount == 0, only supports mandatory B32 SAS
                sasMode = SasType.B32;
                for (int i = 0; i < sasCount; i++) {
                    // Only need to check for B256 as, if its not there,
we'll
                    // always use B32
                    if (platform.getUtils().equals(SasType.B256.getType(), 0,
aMsg,
                            sasPos + i * 4, 4)) {
```

Commented [s21]: Next we negotiate the Auth Tag types, which defaults to HS32 but can also be HS80 (these are short for HMAC-SHA-32 and HMAC-SHA-80, respectively).

Commented [s22]: Now get the Key Agreement Type, which must be DH3K at the minimum. Look for more efficient alternatives below, such as ECDH256 or ECDH384 (these are Elliptic Curve algorithms of varying strengths).

Commented [s23]: Negotiate the Short Authentication String, which is base32-encoded by default, but if the more efficient base256 alternative is available, use that instead.

```
                    sasMode = SasType.B256;
                }
                if (platform.isVerboseLogging()) {
                    logString("HELLO MSG - SAS: "
                            + new String(aMsg, sasPos + i * 4, 4));
                }
            }

                                    ...

            rxHelloMsg = aMsg;
            sendHelloACK();
        }
    }
```

Commented [s24]: Store the “hello” message for later use as input for hashing

Commented [s25]: Send the actual packet with the string “HelloACK”

After an endpoint receives a HelloACK message, one of two things can happen. Either the endpoint can wait for the other party to give the next signal, or the endpoint can send a “Commit” message, which is a signal to begin the key exchange. The message contains the results of the algorithm negotiations, essentially locking both parties into using them. For this reason, the party sending the “Commit” message is called the initiator:

```
private synchronized void sendCommit() throws IOException {
        if (platform.getLogger().isEnabled()) {
            logString("Sending COMMIT...");
        }
                                    * * *

        if (commitMsg == null) {
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            // Commit always has length 29 words in DH mode (which we
always
            // use)
            baos.write(createMessageBase(MSG_TYPE_COMMIT, 29));
            baos.write(hashChain.H2);
            baos.write(localZID);
            byte[] hash = dhMode.hash.getType();
            baos.write(hash); // We only use SHA-256
            baos.write(cipherInUse.getSymbol());
            baos.write(authMode.getSymbol());
            baos.write(dhMode.getType());
            baos.write(SasType.B256.getType());
            baos.write(createHvi());
            byte[] commit = baos.toByteArray();
            baos.close();
            commitMsg = addImplicitHMAC(commit, hashChain.H1);
            dhSuite.setAlgorithm(dhMode);
                                    * * *

        }
        // Save the contents of COMMIT to be sent
        msgCommitTX = commitMsg;
        sendZrtpPacket(commitMsg);
  }
```

Commented [s26]: Checks that we don’t already have a commit message, which may happen due to network issues

Commented [s27]: Here we begin populating the Commit message with all the agreed-upon algorithms

Commented [s28]: This is later used by the receiving party to verify that it matches the Hello / HelloACK messages from before, as a check on integrity.

Commented [s29]: Actually sends the constructed packet

References:

ZRTP: Media Path Key Agreement for Unicast Secure RTP P. Zimmermann, A. Johnston, J. Callas [April 2011] (TXT = 293784) (Status: INFORMATIONAL) (Stream: IETF, WG: NON WORKING GROUP) (DOI: 10.17487/RFC6189)

More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE) T. Kivinen, M. Kojo [May 2003] (TXT = 19166) (Status: PROPOSED STANDARD) (Stream: IETF, Area: sec, WG: ipsec) (DOI: 10.17487/RFC3526)