Documentation of the ZK-SSH Project

Andreas Gaupmann Christian Schausberger Ulrich Zehl

September 7, 2005

Abstract

OpenSSH is an open source implementation of the SSH standard defined by the IETF Secure Shell Working Group. SSH provides a secure way of accessing remote systems. This goal can only be achieved if it incorporates also user authentication which ensures the identity of a user. Several methods like public key, password and hostbased authentication methods are implemented in OpenSSH. Public key systems like RSA and DSA provide a reasonable level of security in regard to properties like transferability and impersonation. Nevertheless, these challenge–response methods leak in polynomial time information to a third party making it easier to impersonate another party. This problem does not exist when a zero–knowlegde user authentication protocol is used. Therefore, a ZK protocol was chosen (Ohta–Okamoto) and implemented for usage with OpenSSH.

Contents

1 SSH–Drafts

1.1 SSH–TRANS

1.1.1 Binary Packet Protocol

According to chapter 6 of SSH–TRANS [YL05c] each packet is in the following format:

```
uint32    packet_length
byte      padding_length
byte[n1]  payload; n1 = packet_length - padding_length - 1
byte[n2]  random padding; n2 = padding_length
byte[m]   mac (Message Authentication Code - MAC); m = mac_length
```

1.1.2 Service Requests

As soon as a secure transport connection is established between server and client (after key exchange) the client requests a service. Currently, two service requests are defined. Their names are:

ssh-userauth: Client user wants to authenticate itself to the server.

ssh-connection: Through an SSH connection different channels can be defined, eg interactive session (shell), X11 forwarding, ...

The payload of a service request is built as follows:

```
byte      SSH_MSG_SERVICE_REQUEST
string    service name
```

If the server accepts the service request then it sends the message

```
byte      SSH_MSG_SERVICE_ACCEPT
string    service name
```

otherwise a disconnect message is sent

```
byte      SSH_MSG_DISCONNECT
uint32    reason code
string    description [RFC3629]
string    language tag [RFC3066]
```

Other possible responses of the server are:

- `SSH_MSG_IGNORE` (against traffic analysis)
- `SSH_MSG_DEBUG`
- `SSH_MSG_UNIMPLEMENTED` (unrecognized packet)

A particularly interesting message for development is the debugging message:

```
byte      SSH_MSG_DEBUG
boolean   always_display
string    message [RFC3629]
string    language tag [RFC3066]
```

1.2 SSH–USERAUTH

1.2.1 General Notes

The user authentication of the SSH protocol is defined in a draft [YL05a] of the IETF Secure Shell Working Group. It depends on the SSH transport layer for secure communication over insecure networks. SSH–TRANS is standardized in a draft document [YL05c]. Without a secure transport layer, providing integrity and confidentiality, the user authentication protocol is not secure.

1.2.2 Authentication Requests

The authentication protocol receives a session identifier from the transport protocol. The session identifier is derived from the key exchange hash value. The server controls the authentication. The server tells the client the user authentication methods it supports. The client can choose one of these methods.

According to SSH–USERAUTH [YL05a] all authentication requests must adhere to the following format:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name in ISO-10646 UTF-8 encoding
string    service name in US-ASCII
string    method name in US-ASCII
The rest of the packet is method-specific.
```

The important part is the last sentence. Every user authentication method can have its own fields.

In the draft there are following user authentication methods defined:

- public key (REQUIRED)
- password (OPTIONAL)
- hostbased (OPTIONAL)
- none (NOT RECOMMENDED)

Additional authentication methods can be defined. The naming must conform to SSH–ARCH [YL05b] and SSH–NUMBERS [LL05].

If the server rejects the authentication request, it responds with the following:

```
byte         SSH_MSG_USERAUTH_FAILURE
name-list    authentications that can continue
boolean      partial success
```

The "authentications that can continue" is a comma–separated name–list of authentication "method name" values that may productively continue the authentication dialog.

When the server accepts authentication, it responds with the following:

```
byte      SSH_MSG_USERAUTH_SUCCESS
```

Note that this is not sent after each step in a multi–method authentication sequence, but only when the authentication is complete.

2 User authentication method "zk"

2.1 The Ohta-Okamoto Zero-Knowledge Identification Protocol

The zero-knowledge protocol used by the ZK-SSH project is due to Ohta and Okamoto [OO88]. It is a modification and generalization of the well-known Fiat-Shamir scheme [FS86], but has one crucial advantage: It is, to the best of our knowledge, not encumbered by patents.

Ohta and Okatomo present a sequential version in some detail, and discuss the security of a parallel scheme, but they do not present the parallel protocol's actions. Therefore, we will do this ourselves.

2.1.1 Setup

A prover P chooses a RSA-like modulus n (the product of two secret large primes) and k random integers $S_i \in \mathbb{Z}_n$ as his private identity keys, and a small integer L. His overall private key is $((S_1, \ldots, S_k), n, L)$.

P now computes his public identity keys $I_i = {S_i}^L$ for all $1 \leq i \leq k$, and publishes $((I_1, \ldots, I_k), n, L)$ as his overall public identity key.

2.1.2 Proof of Identity and Verification

The proving party P wants to prove their purported identity to the verifying party V. To do this, the following protocol is carried out over t rounds until V is convinced of the correctness.

1. P chooses a random $R \in \mathbb{Z}_p$ and sends the witness $X = R^L$ to V.
2. V picks $(e_1, \ldots, e_k) \in \mathbb{Z}_L$ at random, and sends the resulting challenge vector to P.
3. P computes Y as
$$Y = R \cdot \prod_{j=1}^{k} (S_j)^{e_j} \pmod{n}$$
and sends Y as his response.
4. V accepts the round if the following equation holds.
$$Y^L = X \cdot \prod_{j=1}^{k} (I_j)^{e_j} \pmod{n}$$

After t successful rounds, V accepts the proof. The probability that a cheating prover can successfully fool an honest verifier is then only L^{-kt}, a significant gain over the sequential protocol's L^{-t}.

2.2 Concrete Parameters

In the ZK-SSH project, we will work in a remote attacher scenario where no reasonable assumptionsabout his power—beyond being polynomially bounded—are reasonable. Therefore, the probability that a cheater can successfully fool an honest verifier should be
$$\frac{1}{2^{80}}$$
or less.

Chosing $L := 4$, $k := 10$ as the number of secret identitiy keys, and $t := 4$ as the number of rounds and substituting in the respective equations yields

$$\left(\frac{1}{4}\right)^{10 \cdot 4} = \left(\left(\frac{1}{2}\right)^{2}\right)^{40} = \left(\frac{1}{2}\right)^{80}$$

which statisfies the above requirement.

For the length of the modulus n, we propose to use at least 2048 bits to have a margin of security for future developments in factorization and cryptanalysis.

2.3 Method messages

In chapters 7, 8 and 9 of the user authentication draft [YL05a] the methods pubkey, password and hostbased are defined. The new zero knowledge method will be layed out accordingly by defining the used messages and their fields.

For speeding up the protocol and reducing generated overhead of sent packets a parallel version of the Ohta–Okamoto protocol is used in this project.

"zk" is an optional authentication method. Data types are defined in SSH–ARCH [YL05b]. ZK authentication uses the following packets. The client starts by sending a user authentication request.

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name in ISO-10646 UTF-8 encoding
string    service name in US-ASCII
string    "zk"
int       round
mpint     witness (X)
mpint     exponent of user public key (L)
mpint     modulus of user public key (n)
int       number of identities of user public key (k)
mpint     identity 1
mpint     identity 2
.
.
mpint     identity k
```

This message is used to start the first round of a zero–knowledge user authentication dialog. In the paper of Ohta–Okamoto a "round" designates a challenge and a response. In our implementation of the protocol we have chosen to refer to challenges and responses as rounds on their own. Thus, the paramter $L = 4$ from section 2.2 stands for eight rounds. Just imagine we have broken up the four rounds in eight half–rounds.

On receipt of such a message the server extracts the public key and compares it against a list of valid keys. Unless the received public key of the user is contained in the list the user is rejected.

If the server accepts the method "zk" then it replies to the first message with either SSH_MSG_USERAUTH_FAILURE or by issuing a challenge.

```
byte      SSH_MSG_USERAUTH_ZK_OK
int       round
mpint     challenge 1 (e_1)
mpint     challenge 2 (e_2)
.
.
mpint     challenge k (e_k)
```

The client responds with this message:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service
string    "zk"
int       round
mpint     response (Y)
```

When the server receives this message, it checks whether the supplied key is acceptable for authentication, and if so, it checks whether the verification condition holds true:

$$Y^L \equiv \left(R \cdot \prod_{j=1}^{k} S_j^{e_j} \right)^L \equiv R^L \cdot \prod_{j=1}^{k} \left(S_j^L \right)^{e_j} \equiv X \cdot \prod_{j=1}^{k} I_j^{e_j} \bmod n$$

If both checks succeed, this method is successful. The server may require additional authentications. The server responds with SSH_MSG_USERAUTH_SUCCESS (if no more authentications are needed), or SSH_MSG_USERAUTH_FAILURE (if the request failed, or more authentications are needed).

The following method–specific message numbers are used by the "zk" authentication method.

```
SSH_MSG_USERAUTH_ZK_OK              60
```

3 Current implementation of user authentication in OpenSSH 4.0p1

The source code organization of OpenSSH is very flat. All files are contained within the base directory. Subdirectories exist for extensions and patches (contrib), OpenBSD compatability files (openbsd–compat), regression tests (regress)

Table 1: Source code files of server and client

User authentication	**Server**	**Client**
started in	sshd.c	ssh.c
defined in	auth.h	sshconnect.h
implemented in	auth2.c auth2-chall.c auth2-gss.c auth2-hostbased.c auth2-kbdint.c auth2-none.c auth2-passwd.c auth2-pubkey.c	sshconnect2.c
communication with ssh–agent		authfd.c

and smart card handling (scard). The source files implementing user authentication are shown in table 1.

The differences between SSHv1 and SSHv2 have also implications on user authentication. Therefore, many implementation (.c) and definition (.h) files exist with an appended 1 or 2 in their names. SSHv1 is considered insecure and should not be used in practice. Thus, the ZK extension was implemented for SSHv2. Another demarcation line runs through the source files. Some are for the server and the others for the client. All the subsequent descriptions for the extension of the source code are divided into client and server side for SSHv2.

3.1 Server

The main file of the server is **sshd.c**. A server process listens on sockets for incoming connections. Every request is handled in a forked child process. There user authentication takes also place. Following call to the function do_authentication2 starts it:

```
01686         /* perform the key exchange */
01687         /* authenticate user and start session */
01688         if (compat20) {
01689                 do_ssh2_kex();
01690                 do_authentication2(authctxt);
01691         } else {
01692                 do_ssh1_kex();
01693                 do_authentication(authctxt);
01694         }
```

First the applied protocol version is determined. In case of SSHv2 do_ssh2_kex is called and a secure connection is established between server and client. After

establishment of the SSH transport layer user authentication can proceed. The function do_authentication2 is implemented in the source file **auth2.c** and is used only by the server.

```
00087 void
00088 do_authentication2(Authctxt *authctxt)
00089 {
00090         /* challenge-response is implemented via keyboard interactive */
00091         if (options.challenge_response_authentication)
00092                 options.kbd_interactive_authentication = 1;
00093
00094         dispatch_init(&dispatch_protocol_error);
00095         dispatch_set(SSH2_MSG_SERVICE_REQUEST, &input_service_request);
00096         dispatch_run(DISPATCH_BLOCK, &authctxt->success, authctxt);
00097 }
```

The dispatching mechanism is outlined in section 3.1.1. User authentication is handled by the server accordingly to the USERAUTH draft as service requests. The name of this special service request is "ssh–userauth". The function input_service_request calls in case of a ssh–userauth service request the function dispatch_set.

```
00111         if (strcmp(service, "ssh-userauth") == 0) {
00112                 if (!authctxt->success) {
00113                         acceptit = 1;
00114                         /* now we can handle user-auth requests */
00115                         dispatch_set(SSH2_MSG_USERAUTH_REQUEST, &input_userauth_request);
00116                 }
00117         }
```

By the time dispatch_run runs a callback function (input_userauth_request) is already registered for handling this service request (SSH2_MSG_USERAUTH_REQUEST). That function is also implemented in **auth2.c**.

```
00132 static void
00133 input_userauth_request(int type, u_int32_t seq, void *ctxt)
```

In this function one code construct is used for handling all user authentication methods. The currently conducted method is stored in following structure:

```
00136         Authmethod *m = NULL;
```

Later the chosen method is started to run by dereferencing the member "userauth" of the Authmethod struct m (see section 3.1.2 for an explanation of the structures used by the server). The return value of the callback function is either 0 (FALSE) or 1 (TRUE) and is stored in the variable authenticated:

```
00196         /* try to authenticate user */
00197         m = authmethod_lookup(method);
00198         if (m != NULL) {
00199                 debug2("input_userauth_request: try method %s", method);
00200                 authenticated = m->userauth(authctxt);
00201         }
```

Thus, for every user authentication method a different callback function gets called. These functions are listed in section 3.1.3.

3.1.1 Dispatching (dispatch.c)

A callback mechanism has been implemented in OpenSSH for processing incoming service requests. In a timely manner callback functions are registered for the expected packets just before they arrive. The same is true for deleting them as soon as they are not needed any more. Pointers to callback functions are stored in the array dispatch.

```
00037 dispatch_fn *dispatch[DISPATCH_MAX];
```

The array is declared for holding DISPATCH_MAX elements. The elements are of type pointer to dispatch_fn (fn stands for function). dispatch_fn is defined in the file dispatch.h:

```
00031 typedef void dispatch_fn(int, u_int32_t, void *);
```

The typedef defines a new type dispatch_fn which is a function taking three parameters of the types shown above and returns void. The dispatching mechanism consists of three major functions:

- dispatch_init
- dispatch_set
- dispatch_run

First the function dispatch_init is called for initializing all elements of the dispatch array with a default message handler (dispatch_protocol_error). Thus, if a wrong element is accessed the error can be handled instead of throwing a segmentation fault. A pointer to this error handler is passed to dispatch_init:

```
00055 void
00056 dispatch_init(dispatch_fn *dflt)
00057 {
```

```
00058         u_int i;
00059         for (i = 0; i < DISPATCH_MAX; i++)
00060                 dispatch[i] = dflt;
00061 }
```

Registering callbacks is done with the function dispatch_set. The message type is used as index into the array. The element at this index is used for storing a pointer to the callback function for handling this specific type of message.

```
00073 void
00074 dispatch_set(int type, dispatch_fn *fn)
00075 {
00076         dispatch[type] = fn;
00077 }
```

The callback function itself is called in the function dispatch_run. In it an infinite loop is entered and finally the callback function gets called. The loop is only left if the user authentication method succeeds.

```
00092                 if (type > 0 && type < DISPATCH_MAX && dispatch[type] != NULL)
00093                         (*dispatch[type])(type, seqnr, ctxt);
```

Line 93 should be read as follows: The element at position type of the array dispatch is dereferenced. In that case the element is a pointer to a function. Therefore, this function is called. The variables type, seqnr and ctxt are passed to the function. If the type is SSH2_MSG_USERAUTH_REQUEST than the function input_userauth_request is dereferenced which is defined in **auth2.c**:

```
00132 static void
00133 input_userauth_request(int type, u_int32_t seq, void *ctxt)
```

The logic behind user authentication on the server (dispatching of the right authentication method) is in the file **auth2.c**. The actual implementation of the different user authentication methods is deferred to the files

- auth2-chall.c
- auth2-gss.c
- auth2-hostbased.c
- auth2-kbdint.c
- auth2-none.c
- auth2-passwd.c
- auth2-pubkey.c

3.1.2 Authctxt and Authmethod structures on the server

Defined options and received data from the client regarding user authentication are stored in an authentication structure. The Authctxt struct is defined in the file **auth.h**.

```
0049 struct Authctxt {
0050    int      success;
0051    int      postponed; /* authentication needs another step */
0052    int      valid;     /* user exists and is allowed to login */
0053    int      attempt;
0054    int      failures;
0055    int      force_pwchange;
0056    char        *user;      /* username sent by the client */
0057    char        *service;
0058    struct passwd   *pw;        /* set if 'valid' */
0059    char        *style;
0060    void        *kbdintctxt;
0061 #ifdef BSD_AUTH
0062    auth_session_t  *as;
0063 #endif
0064 #ifdef KRB5
0065    krb5_context     krb5_ctx;
0066    krb5_ccache  krb5_fwd_ccache;
0067    krb5_principal   krb5_user;
0068    char        *krb5_ticket_file;
0069    char        *krb5_ccname;
0070 #endif
0071    void        *methoddata;
0072 };
```

Every method for user authentication is described by a structure called Authmethod. This struct is defined in the file **auth.h**:

```
00080 struct Authmethod {
00081         char    *name;
00082         int     (*userauth)(Authctxt *authctxt);
00083         int     *enabled;
00084 };
```

The members name and enabled are straight forward. The first holds a textual identifier for the method and the latter is a flag indicating if this method is allowed to be used. userauth is a pointer to a function that takes a pointer to a Authctxt struct and returns an integer. If 1 (TRUE) is returned then the user authentication was successful; in case of 0 (FALSE) that method has failed.

Table 2: User authentication methods on the server

Method	Source file	Note
none	auth2–none.c	querying list of methods from server
pubkey	auth2–pubkey.c	
passwd	auth2–passwd.c	PAM can be used
kbdint	auth2–kbdint.c, auth2–chall.c	password, SecurID, PAM
hostbased	auth2–hostbased.c	
gssapi	auth2–gss.c	Kerberos

3.1.3 User authentication functions on the server

For every supported method for user authentication there exists a source file. Challenge–response is currently implemented via keyboard–interactive in OpenSSH. Table 2 gives an overview.

In every source code file listed in table 2 an Authmethod struct is initialized and a function for handling the authentication process is defined:

- `userauth_passwd(Authctxt *authctxt)`
- `userauth_pubkey(Authctxt *authctxt)`
- `userauth_kbdint(Authctxt *authctxt)`
- `userauth_none(Authctxt *authctxt)`
- `userauth_hostbased(Authctxt *authctxt)`
- `userauth_gssapi(Authctxt *authctxt)`

Let m be a variable of type Authmethod. If an expression like `m->userauth(authctxt)` is executed then one of the above listed functions gets called.

3.2 Client

The main source file for the client is **ssh.c**. In this file the login to a SSH–Server is started by calling the function in the line given below. The client builds a secure transport connection to the server; then user authentication is conducted.

```
00703         /* Log into the remote system.  This never
                 returns if the login fails. */
00704         ssh_login(&sensitive_data, host,
                 (struct sockaddr *)&hostaddr, pw);
```

After key exchange (kex) between client and server has finished, a secure transport layer connection is established between client and server. ssh_login is defined in **sshconnect.c**:

```
00938 ssh_login(Sensitive *sensitive, const char *orighost,
00939     struct sockaddr *hostaddr, struct passwd *pw)
```

At this point the differentiation between SSHv1 and SSHv2 occurs on the client.

```
00959         /* key exchange */
00960         /* authenticate user */
00961         if (compat20) {
00962                 ssh_kex2(host, hostaddr);
00963                 ssh_userauth2(local_user, server_user, host, sensitive);
00964         } else {
00965                 ssh_kex(host, hostaddr);
00966                 ssh_userauth1(local_user, server_user, host, sensitive);
00967         }
```

The function `ssh_userauth2` is implemented in the source file **sshconnect2.c**:

```
00255 ssh_userauth2(const char *local_user, const char *server_user, char *host,
00256     Sensitive *sensitive)
```

If the option for challenge-response is enabled then keyboard-interactive is used. Then the initial userauth request is assembled. This is a message of type SSH2_MSG_SERVICE_REQUEST. The requested service is called "ssh–userauth". The user authentication proceeds if the server sends back a SSH2_MSG_SERVICE_ACCEPT. In this case the client sets up a new authentication context. In this context the initial user authentication method is set to "none". From this context a user authentication request is built and sent to the server. Usually the server rejects method "none" and sends back a list of valid methods to the client. The client tries now method by method until one or more or none succeeds. For handling the answers from the server during these attempts callback functions are registered for dispatching as explained in section 3.1.1.

```
00304         dispatch_init(&input_userauth_error);
00305         dispatch_set(SSH2_MSG_USERAUTH_SUCCESS, &input_userauth_success);
00306         dispatch_set(SSH2_MSG_USERAUTH_FAILURE, &input_userauth_failure);
00307         dispatch_set(SSH2_MSG_USERAUTH_BANNER, &input_userauth_banner);
00308         dispatch_run(DISPATCH_BLOCK, &authctxt.success, &authctxt);     /* loop until success
```

Dispatching works exactly like at the server by using the code in **dispatch.c**. If a method fails then the control flow goes to the function input_userauth_failure. In this function a list of valid methods is read from the server response and calls the function userauth. The list of user authentication methods is passed to this function.

```
00316 void
00317 userauth(Authctxt *authctxt, char *authlist)
```

In this function an infinite loop is entered:

```
00336                /* reset the per method handler */
00337                dispatch_range(SSH2_MSG_USERAUTH_PER_METHOD_MIN,
00338                    SSH2_MSG_USERAUTH_PER_METHOD_MAX, NULL);
00339
00340                /* and try new method */
00341                if (method->userauth(authctxt) != 0) {
00342                        debug2("we sent a %s packet, wait for reply", method->name);
00343                        break;
00344                } else {
00345                        debug2("we did not send a packet, disable method");
00346                        method->enabled = NULL;
00347                }
```

In line 341 the member "userauth" of the structure method is called (see section 3.2.1). It is a pointer to a function and should not be confused with the function userauth mentioned above. Dereferencing this pointer leads to calling one of the functions listed in section 3.2.2. Therefore, a different callback function is called for every user authentication method.

The function dispatch_range clears the elements in the range given by the first two parameters with the value of the third, in this case NULL. Thus, all callbacks regarding user authentication are deregistered.

3.2.1 Authctxt and Authmethod structures on the client

Authentication contexts (Authctxt) are used on both sides (client and server). The structures defining the authentication contexts although are different. Differing authentication method structures (Authmethod) are also used by client and server. Authctxt and Authmethod used by the client are defined in the file **sshconnect2.c**.

```
00164 struct Authctxt {
00165    const char *server_user;
00166    const char *local_user;
00167    const char *host;
00168    const char *service;
00169    Authmethod *method;
00170    int success;
00171    char *authlist;
00172    /* pubkey */
00173    Idlist keys;
00174    AuthenticationConnection *agent;
00175    /* hostbased */
00176    Sensitive *sensitive;
00177    /* kbd-interactive */
00178    int info_req_seen;
```

```
00179    /* generic */
00180    void *methoddata;
00181 };
00182 struct Authmethod {
00183    char    *name;      /* string to compare against server's list */
00184    int (*userauth)(Authctxt *authctxt);
00185    int *enabled;   /* flag in option struct that enables method */
00186    int *batch_flag;    /* flag in option struct that disables method */
00187 };
```

3.2.2 User authentication functions on the client

The member "userauth" of the Authmethod struct is a pointer to a function for authenticating the user to the server. The functions are implemented in the file **sshconnect2.c**:

```
00197 int userauth_none(Authctxt *);
00198 int userauth_pubkey(Authctxt *);
00199 int userauth_passwd(Authctxt *);
00200 int userauth_kbdint(Authctxt *);
00201 int userauth_hostbased(Authctxt *);
00202 int userauth_kerberos(Authctxt *);
00205 int userauth_gssapi(Authctxt *authctxt);
```

4 Adjustments to the OpenSSH 4.0p1 code base

In this section the extension of the code base will be outlined. The rules for this undertaking are the SSH drafts published by the IETF Secure Shell Working Group and the coding guidelines of the OpenSSH developers. Furthermore, the format of the extension is a patch (generated with `diff`) that when applied to the OpenSSH portable source code will generate fully functional SSH programs with an additional zero–knowledge user authentication method.

In section 2.3 the messages for ZK authentication were defined. A more demanding task is the actual extension of the OpenSSH source code for generating, sending and processing them.

4.1 Server

4.1.1 Definition of new authentication method "zk"

A new source code file called **auth2–zk.c** was introduced. We found it wise to use the other method implementations as template for ZK. One important point

is to know which header files are used by other user authentication methods and the purposes they serve. Table 3 gives an overview about this matter.

The determination of the required header files for **auth2–zk.c** led to the following list:

- includes.h
- xmalloc.h
- packet.h
- log.h
- key.h
- auth.h
- monitor_wrap.h
- servconf.h
- ssh2.h
- uidswap.h
- openssl/bn.h
- ohta-okamoto.h

The header files not described in table 3 are `openssl/bn.h` and `ohta-okamoto.h`. The first on provides an interface into the OpenSSL cryptographic library for arithmetic with multiple–precision integers. The latter gives access to the actual zero–knowledge protocol implementation.

A function userauth_zk was implemented for answering the requests sent by the client.

```
00115    static int
00116    userauth_zk(Authctxt *authctxt)
```

In the first round the client sends its public key along with the witness to the server. This request is answered by userauth_zk with with a challenge. If the public key is not found in the file `authorized_keys` or `authorized_keys2` then the client gets rejected. Otherwise, the response of the client is afterwards received and checked for its correctness. This sequence of actions is repeated four times to reach an appropriate level of security.

An Authmethod method_zk structure has to be defined to specify the properties of the new ZK user authentication method.

Table 3: Header files used in userauth and their purposes (partially taken from OVERVIEW file)

Header file	**Purpose**
atomicio.h	Ensure all of data on socket comes through.
auth–options.h	Accessing environment variables and parsing of options.
auth.h	Definitions of Authctxt, Authmethod and functions for user authentication.
bufaux.h	see buffer.h
buffer.h	Buffer manipulation routines: These provide an arbitrary size buffer, where data can be appended. Data can be consumed from either end. The code is used heavily throughout ssh. The basic buffer manipulation functions are in buffer.c (header buffer.h), and additional code to manipulate specific data types is in bufaux.c.
canohost.h	Code in canohost.c is used to retrieve the canonical host name of the remote host.
compat.h	Differentiation between SSHv1 and SSHv2 (compat13, compat20, datafellows).
dispatch.h	Definitions for registering and deleting callback functions for the handling of client requests.
includes.h	Includes most system headers. Lots of `#ifdef`s.
key.h	Interface to the OpenSSL RSA and DSA key handling routines.
log.h	The implementation that logs to system log is in log.c; the definitions are in log.h.
misc.h	String manipulation, processing of variable–length argument lists and querying passphrases.
monitor_wrap.h	Functions called in case of privilege separation (names begin with `mm_`).
packet.h	Binary packet protocol: • The ssh binary packet protocol is implemented in packet.c. The code in packet.c does not concern itself with packet types or their execution; it contains code to build packets, to receive them and extract data from them, and the code to compress and/or encrypt packets. CRC code comes from crc32.c. • The code in packet.c calls the buffer manipulation routines (buffer.c, bufaux.c), compression routines (compress.c, zlib), and the encryption routines.
pathnames.h	Definition of paths to various configuration, executable and key files.
servconf.h	Reading the configuration file (servconf.c) of the server.
ssh.h	The main header file for ssh (various definitions).
ssh2.h	Definitions of the message types used in SSHv2 according to SSH–ARCH [YL05b].
ssh–gss.h	Interface to GSS–API and definitions of message types, structures and functions for Kerberos authentication.
uidswap.h	uid–swapping
xmalloc.h	"safe" malloc routines

```
Authmethod method_zk = {
    "zk",                       /* name */
    userauth_zk,                /* userauth*/
    &options.zk_authentication  /* enabled */
};
```

To define whether ZK method is enabled or not the command line options and the configuration file statements were extended to allow for ZK user authentication (see section 4.1.2 for details).

Sending a packet can be achieved by using the interface for the Binary Packet Protocol provided through **packet.h**. The process of sending an answer to a service request for user authentication using ZK is outlined below.

```
00183 /* send challenge to client */
00184 packet_start(SSH2_MSG_USERAUTH_ZK_OK);
00185 packet_put_int(round);
00186 for (i=0; i<public.num; ++i)
00187 packet_put_bignum2(challenge[i]);
00188 packet_send();
00189 packet_write_wait();
00190
00191 /* 8 rounds; only in last round will be decided if
   user is authenticated successfully */
00192 authctxt->postponed = 1;    /* wait for client answer */
```

The nifty details of assembling the packet that really goes out on the wire is hidden in comfortable high–level functions. Therefore, the building of custom packet layouts like that for ZK authentication messages is achieved easily. First comes the message type, then the number of the (half) round is written. That is followed by a fixed number of challenges. Subsequently, the packet is sent and the server is told to wait for an answer of the client. Meanwhile the authentication dialog is postponed until this answer arrives.

The authentication options are parsed in **ssh.c** and the server configuration file is read in **servconf.c**. The default options are also set in this file. Obviously, ZK user authentication is enabled by default along with public key and password methods.

4.1.2 New options and configuration settings

The extension of OpenSSH with a zero knowledge user authentication has led to the introduction of a new option in **sshd_config**. By defining

```
ZeroKnowledgeAuthentication yes
```

the user can enable zero knowledge user authentication. By default ZK authentication will be used as one of the preferred user authentication methods (the others are "public key" and "keyboard–interactive"). Zero knowledge authentication can be disabled explicitly by passing `no` to the above shown option.

4.1.3 Privilege separation

Privilege separation is an optional mode of running for the server. It increases security by reducing the runtime in which the server possesses user or even root privileges. This extra privileges are only needed for binding to port 22 or reading keys. The rest of the code can be executed without the need to use them. For some reason the ZK protocol implementation does not function when privilege separation is enabled. The reason for this is unknown yet.

4.1.4 Modified files

4.1.5 New files

- **auth2–zk.c**

4.2 Client

4.2.1 sshconnect2.c, the main client file

In contrast to the server where every user authentication method is implemented in its own implementation file, at the client side all methods are joined in the file **sshconnect2.c**. In this file exists an array of authentication methods and one or more functions specifically for handling a certain method. The array has to be extended by one element and at least two new functions have to be introduced:

```
Authmethod authmethods[] = {
    {"zk",                              /* name */
      userauth_zk,                      /* userauth */
      &options.zk_authentication,       /* enabled */
      NULL},                            /* batch_flag */
    {"hostbased",
      userauth_hostbased,
      &options.hostbased_authentication,
      NULL},
    .
    .
    .
}
```

```
int
userauth_zk(Authctxt *authctxt);

void
input_userauth_zk_ok(int type, u_int32_t seq, void *ctxt);
```

The last two functions send the initial ZK userauth request and the answer to a SSH_MSG_USERAUTH_ZK_OK received from the server, respectively. These messages could be assembled in a manner similar to this one:

```
packet_start(SSH2_MSG_USERAUTH_REQUEST);
packet_put_cstring(authctxt->server_user);
packet_put_cstring(authctxt->service);
packet_put_cstring(authctxt->method->name);
packet_put_char(0);
packet_put_bignum(n);
packet_put_bignum(I);
packet_put_bignum(X);
packet_add_padding(64);
packet_send();
```

4.2.2 New options and configuration settings

The extension of OpenSSH with a zero knowledge user authentication has led to the introduction of a new option in **ssh_config**. By defining

```
ZeroKnowledgeAuthentication yes
```

the user can enable zero knowledge user authentication. By default ZK authentication will be used as one of the preferred user authentication methods (the others are "public key" and "keyboard–interactive"). Zero knowledge authentication can be disabled explicitly by passing `no` to the above shown option.

4.2.3 Modified files

- **ssh.c**
- **sshconnect2.c**

4.2.4 New files

No new files were added to the client part of the code base.

4.3 Key handling

4.3.1 General

The key handling (accessing, reading and freeing of keys) in OpenSSH is not intuitive and partly just messy. Therefore, a short overview of that matter shall be given in this section.

Traditional authentication methods that use public key cryptography are:

hostbased: No user authentication takes place. Instead only the host key of the connecting client is checked by the server. If the validity of the client's public key can be verified by the server, access is granted to the client.

pubkey: Here the key pair belongs to the user. RSA and DSA keys can be generated with `ssh-keygen`. The public key has to be copied to the user's authorized keys file on the server. Otherwise, the public key of the user can not be verified by the server.

The newly implemented zero–knowledge user authentication also belongs into the same category.

There exist several source code files that implement the creation and manipulation of keys. The functions provided by these files are used by server, client and key generator programs.

key.{c,h}: The header file **ohta-okamoto.h** was included for using the therein defined data types. The enumeration of existing key types (KEY_RSA1, KEY_RSA, KEY_DSA and KEY_UNSPEC) was extended by the entry KEY_OO_ZK. A new data type OOZK for representing a zero–knowledge key pair was introduced. In the structure `Key` a new element `oozk` of type pointer to `OOZK` was added. Therefore, accessing a ZK key throughout the OpenSSH source code could be achieved by adapting already existing functions for key handling to recognize the new key type.

authfile.{c,h}: Provides functions to read (load) and write keys into files on disk. Keys in the RSA1 key format (RSA1 and OOZK) are stored in a binary file directly by OpenSSH. RSA and DSA keys are stored in PEM format by using the OpenSSL library. Several functions were adjusted to handle ZK keys.

4.3.2 Generation and representation of keys

Keys are generated with a separate program named "ssh-keygen". It enables the user to generate RSA1, RSA, DSA and OO_ZK keys and to modify their optional attributes. The generated keys are stored in identity files which are

stored in a `.ssh` subdirectory in the user's home directory. By default following names are used depending on the generated key type:

RSA1: identity

RSA: id_rsa

DSA: id_dsa

OO_ZK: id_oo_zk

Only the private key is stored in this identity files. The public key is written to a file with the same name plus the extension `.pub` appended.

OpenSSH uses the OpenSSL–API functions for PEM encoding to store RSA and DSA keys in files. This is easily done because the RSA and DSA data types are also taken from OpenSSL which provides routines for PEM encoding and decoding exactly for those key (data) types. It offers no type for ZK keys. The same is true for RSA1 keys. Therefore, these two key types are written to disk by using a custom binary file format. For the ZK implementation the RSA1 functions ware taken as a template and adapted to handle OO_ZK keys.

`ssh--kegen` offers beside key generation also options for key manipulation like changing comments for RSA1–style keys and changing the passphrase used for encrypting the private key. This functionality is not implemented for OO_ZK keys due to lacking importance to the actual ZK protocol implementation.

4.3.3 Defining and reading key files

A file containing the public and private keys of a user is called identity file. The default names of these identity files are defined in the file **pathnames.h**:

```
0071 /*
0072 * Name of the default file containing client-side authentication key. This
0073 * file should only be readable by the user him/herself.
0074 */
0075 #define _PATH_SSH_CLIENT_IDENTITY ".ssh/identity"
0076 #define _PATH_SSH_CLIENT_ID_DSA ".ssh/id_dsa"
0077 #define _PATH_SSH_CLIENT_ID_RSA ".ssh/id_rsa"
0078 #define _PATH_SSH_CLIENT_ID_OO_ZK ".ssh/id_oo_zk"
```

The file `.ssh/identity` is used for RSA–rhosts authentication which in OpenSSH equals hostbased authentication. `id_dsa`, `id_rsa` and `id_oo_zk` contain the public and private keys of a user.

The user can define own identity files in the system–wide or per–user configuration file (ssh_config) or by supplying a command line (-i) switch to the `ssh`

command. The two configuration files are read in **readconf.c** for the client and the command line options are parsed in the **ssh.c** source file. Regardless of how the identity files are defined, their names will be stored in a string array in a structure `options` of type `Options`. This functionality is implemented in **ssh.c** in the function `load_public_identity_files`. The function name is misleading because in an identity file, both, the private and public keys of a user are stored. In the function only the public key is read and its name and type are printed as debug output.

The server configuration (including key files) is defined in `sshd_config` or on the command line by the administrator of the server. Default options are set in **servconf.h**. The configuration file and command line options are parsed in **sshd.c**. At startup of the server it reads the server key that is sent to the client before key exchange starts for verifying the authenticity of the server. Later on, in case the client uses a public key method, the server reads valid public keys from the files **authorized_keys** and **authorized_keys2**.

The keys read from file at runtime are stored in a structure `Key` defined in **key.h**:

```
0058 struct Key {
0059    int  type;
0060    int  flags;
0061    RSA *rsa;
0062    DSA *dsa;
0063    OOZK *oozk;
0064 };
```

The type of the key can either be RSA1, RSA, DSA, OO_ZK or UNSPEC. The flags indicate if the key is valid or not. Not all of the references are initialized. A key can only be of one of the mentioned types, e.g. if the key is of type OO_ZK then the pointers `rsa` and `dsa` are not set to valid values. Initially, they are explicitly set to NULL because this condition can be safely checked without risking a segmentation fault.

If the key is not needed anymore then its private part is not only freed but also overwritten. This ensures that no sensitive information is left readable to other processes and subsequently other users. An enhanced function called `xfree` was implemented by the OpenSSH developers for this purpose and is also used throughout the code of the ZK implementation.

The identity files are accessed by the client as soon as zero knowledge user authentication or any method that uses public keys is started. The private key is needed for calculating the witness or signature, respectively. This is conducted in the method–specific functions implemented in the **sshconnect2.c** file.

References

[FS86] Amos Fiat and Adi Shamir, *How to prove yourself: Practical solutions to identification and signature problems*, Advances in Cryptology – CRYPTO 1986 (Andrew M. Odlyzko, ed.), Lecture Notes in Computer Science, vol. 263, Springer, 1986, pp. 186–194.

[LL05] S. Lehtinen and C. Lonvick, *SSH protocol assigned numbers*, URL, IETF Secure Shell Working Group, March 2005, *http://www.ietf.org/internet-drafts/draft-ietf-secsh-assignednumbers-12.txt*.

[OO88] Kazuo Ohta and Tatsuaki Okamoto, *A modification of the Fiat-Shamir scheme*, Advances in Cryptology – CRYPTO 1988 (Shafi Goldwasser, ed.), Lecture Notes in Computer Science, vol. 403, Springer, 1988, pp. 232–243.

[YL05a] T. Ylonen and C. Lonvick, *SSH authentication protocol*, URL, IETF Secure Shell Working Group, March 2005, *http://www.ietf.org/internet-drafts/draft-ietf-secsh-userauth-27.txt*.

[YL05b] ———, *SSH protocol architecture*, URL, IETF Secure Shell Working Group, March 2005, *http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-22.txt*.

[YL05c] ———, *SSH transport layer protocol*, URL, IETF Secure Shell Working Group, March 2005, *http://www.ietf.org/internet-drafts/draft-ietf-secsh-transport-24.txt*.