CS6053 - Network Security

Stack Attack

April 18, 2014

Contents

List of Figures

List of Tables

1 Introduction

The purpose of this lab was to create a program on the client end that connects to a server. On the server end is another program that is running called the `Monitor`. The `Monitor` listens for an open socket on an open port for incoming connections. As long as the `Monitor` is able to authenticate the client, the `Monitor` will reward points. The group with the highest points wins the contest. The challenging part of the contest is to remain a secure connection between the client and the server so each group is rewarded with points. In order to maintain a secure connection we have implemented a few algorithms: Diffie-Hellman, Karn Symmetric Cyrptosystem, and the Fiat-Shamir Algorithm.

2 Implementation

2.1 Authentication of Monitor

In order to authenticate with the `Monitor` we first needed to implement a shared secret key. We did this by implementing the Diffie-Hellman protocol. The Diffie-Hellman key exchange works as shown below:

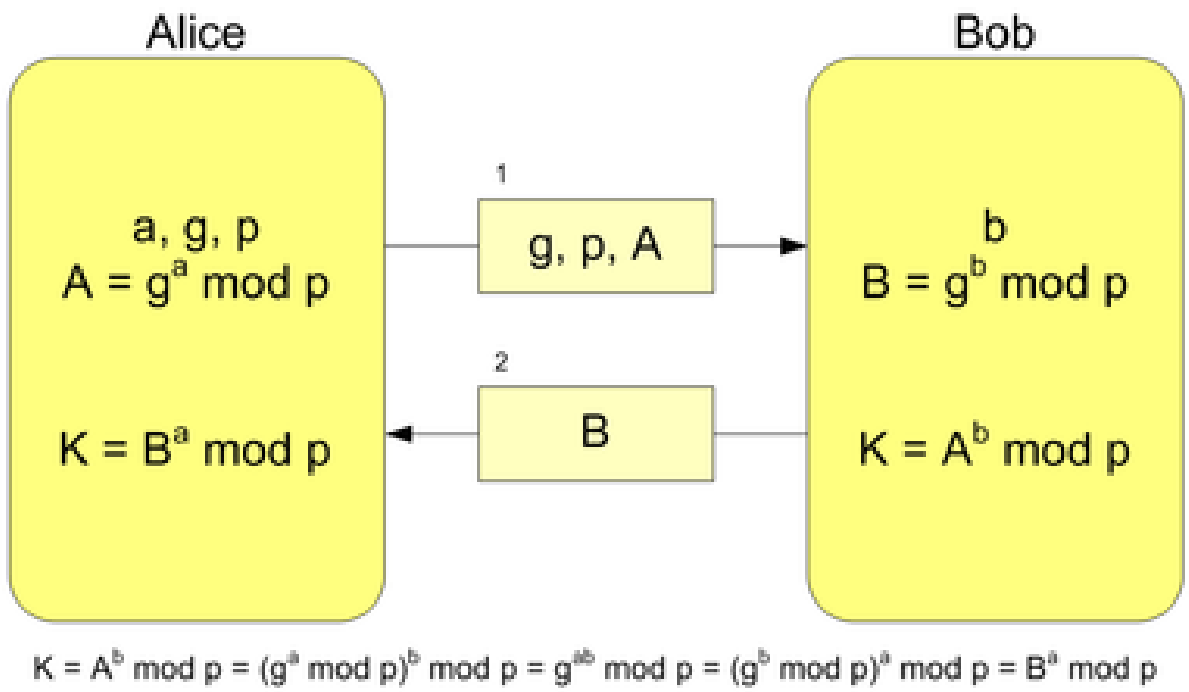

Figure 1: Diffie-Hellman Key Exchange

In the image above, we can see that `g,p` are just the generator and the modulo prime number respectively. `A` is calculated using the function, `K` (g^a mod p); this is then sent over to `Bob`. Note that `g`,`p`, and `A` are all seen by everyone. The trick here is that `Alice` and `Bob` both generate there own secret key locally (this key is never sent across a network). The way we can authenticate can be shown in the example below:

$$
\begin{aligned}
A &= g^a \; mod \; p, \\
B &= g^b \; mod \; p, \\
A &= 3^{15} \; mod \; 17 \equiv 6 \\
B &= 3^{13} \; mod \; 17 \equiv 12
\end{aligned}
$$

where a = 15 (Alice's secret key), g = 3, p = 17 and
where b = 13 (Bob's secret key), g = 3, p = 17

Therefore, `Bob` receives this value from `Alice` along with `p` and `g`. Now `Bob` computes his result and sends `12` to `Alice`. Now the heart of the trick lies here: `Alice` will take Bob's key and use her private key to get the correct message from `Bob` and `Bob` does the same with `Alice's` key:

$$A = 12^{15} \ mod \ 17 \equiv 10$$
$$B = 6^{13} \ mod \ 17 \equiv 10$$

The reason this works is because when you take an exponent to another exponent, these values get multiplied:

$$3^{13^{15}} \ mod \ 17 \equiv 3^{15^{13}} \ mod \ 17$$

Therefore, even if Eve performed a man-in-the-middle attack, all Eve would have is just the encrypted message and the prime and the generator number–making it rather difficult to decrypt the message via brute-forcing due to the discrete logarithm problem.

2.2 Karn Symmetric Cryptosystem

After implementing the Diffie-Hellman protocol we then needed to implement the Karn encryption scheme. The Karn encryption scheme uses the Secure Hash Algorithm (SHA-1) in order to generate the message digests. This algorithm uses the shared keys generated by the Diffie-Hellman protocol. It initially splits the bits of the shared secret key into two halves: a left and a right half. The encryption algorithm works as described below:

- Add "guard" byte of value 42,
- During each clock of plaintext, divide block of plaintext into left/right half,
- Find the message digest of the concatenation of the left plaintext half and the left key half,
- XOR the digest with the right plaintext half. (This produces the right ciphertext half),
- Find the message digest of the ciphertext right half concatenated with the key right half,
- XOR the digest with the plaintext left half.

Once this is complete we have both halves, allowing us to output both the left and right half of the ciphertext.

In order to decrypt the message we needed to follow the steps shown below:

- Strip guard byte,
- For each ciphertext block, split the block into a left half and a right half,
- Find the message digest of the ciphertext right half concatenated with the key right half,
- XOR the result with the ciphertext left half to obtain the plaintext left half,
- Find the message digest of the plaintext left half and the key left half,
- XOR the result with the ciphertext right half to get the plaintext right half.

2.3 Zero-Knowledge Proof

A zero-knowledge proof is a proof such that it will prove a given statement without giving out the "answer." It only gives out the "postulates/theorems" to convince the server that the statement is true without having the server learn anything about the given statement. One example of a zero-knowledge proof is the Fiat-Shamir heuristic algorithm. A simple example can be descibed below:

`Alice` (prover) will not need to have a shared key with `Bob` (verifier); all Alice needs to do is convince `Bob` that this is truly her without `Bob` having the key. `Alice` will prove the knowledge of the secret to `Bob` in `t` executions. The probability is found to be 2^{-t}. Therefore, the more iterations `Alice` tries to prove to `Bob`, the harder it is for an attacker to "impersonate" `Alice`.

The protocol can be described below:

- Initially, `A` will select two primes `p,q`, and multiply them together to generate the public key (n).
- `A` will then select `s` coprime to `n`, where $1 \leq s \leq$,
- `A` will then compute `v`:

$$v \equiv s^2 \ mod \ n$$

- `A` chooses random commitment `r`, $1 \leq$ `r` $\leq$ `n-1`
- `A` sends `B`:

$$x \equiv r^2 \ mod \ n$$

- `B` then sends `A` a random `e`, {0,1}
- `A` sends `B`:

$$y \equiv r * s^e \ mod \ n$$

Once `A` has tried to prove to `B` that this is the correct user, `B` must verify. The verification process is shown below:

- `B` rejects if y = 0,
- `B` accepts if:

$$y^2 \equiv x * v^e \ mod \ n,$$
$$rejects \ otherwise$$

2.4 Maintaining Security

In order to maintain security, we implemented the algorithms described above.

3 Appendix

3.1 Python Code

```python
import sys
import socket
import argparse
from printer import Printer
from config import Config
from diffie_hellman import DHE
from karn import Karn
from fiat import Prover
from base32 import base32
import parse_log

config = Config()
printer = Printer("client")
dhe = DHE()
prover = Prover()
authenticated = False
karn = None
transfer = None
exit = False
only_do_alive = False

def generate_response(line):
    global karn
    global authenticated
    global transfer
    global exit

    line = line.strip()
    directive, args = [x.strip() for x in line.split(':', 1)]

    if directive == "REQUIRE":
        if args == "IDENT":              return "IDENT %s %s" % (Config.ident, dhe.public_key)
        elif args == "PASSWORD":         return "PASSWORD %s" % Config.password
        elif args == "ALIVE":            return "ALIVE %s" % Config.cookie
        elif args == "HOST_PORT":        return "HOST_PORT %s %s" % (Config.server_ip, Config.server_port)
        elif args == "PUBLIC_KEY":       return "PUBLIC_KEY %s %s" % (base32(prover.v), base32(prover.n))
        elif args == "AUTHORIZE_SET":    return "AUTHORIZE_SET %s" % prover.authorize_set()
        elif args == "SUBSET_J":         return "SUBSET_J %s" % prover.subset_j()
        elif args == "SUBSET_K":         return "SUBSET_K %s" % prover.subset_k()
        else:                            printer.error("Unknown require: " + line)
    elif directive == "RESULT":
        args = args.split(' ', 1)
        if args[0] == "IDENT":
            dhe.monitor_key(args[1])
            karn = Karn(dhe.secret)
            printer.info("Setup secret key")
        elif args[0] == "PASSWORD":
            Config.cookie = args[1]
            printer.info("Got cookie: " + Config.cookie)
        elif args[0] == "HOST_PORT":
            printer.info("Login successful! (%s)" % args[1])
            if transfer is None and not manual_mode:
                exit = True
                return ""
        elif args[0] == "ALIVE" and args[1] == "Identity has been verified.":
            printer.info("Alive verified")
            authenticated = True
            if only_do_alive:
                exit = True
                return ""
        elif args[0] == "TRANSFER_REQUEST":
            printer.info("Transfer was %s" % args[1])
        elif args[0] == "ROUNDS":
            prover.rounds = int(args[1])
        elif args[0] == "SUBSET_A":
            prover.subset_a = [int(x) for x in args[1].split()]
        elif args[0] == "TRANSFER_RESPONSE" and not manual_mode:
            exit = True
            return ""
        else:
            printer.error("Unknown result: %s (args: %r)" % (line, args))
    elif directive == "WAITING":
        if transfer and authenticated:
            rt = "TRANSFER_REQUEST %s %s FROM %s\n" % tuple(transfer)
            transfer = None
```

```
            return rt
        if manual_mode and authenticated:
            return raw_input('Command: ')
    elif directive == "COMMENT":
        pass
    elif directive == "COMMAND_ERROR":
        exit = True
        printer.error(line)
        return ""
    else:
        exit = True
        printer.error("Unknown directive: " + line)
        return ""

    return None

if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument('--ident', default="mtest16")
    parser.add_argument('--transfer', nargs=3, metavar=('TO', 'AMOUNT', 'FROM'))
    parser.add_argument('--manual', action='store_true')
    parser.add_argument('--alive', action='store_true')
    parser.add_argument('--logfile', default="/home/httpd/html/final.log.8180")
    args = parser.parse_args()

    manual_mode = args.manual
    transfer = args.transfer
    only_do_alive = args.alive

    if args.ident not in Config.accounts:
        print "Invalid ident"
        sys.exit(1)

    Config.ident = args.ident
    Config.server_port = Config.accounts[args.ident].port

    parse_log.parse(args.logfile)
    Config.cookie = parse_log.cookies[args.ident.lower()]
    Config.password = parse_log.passes[args.ident.lower()]

    print "Using cookie: '%s'" % Config.cookie

    sock = socket.create_connection((Config.monitor_ip, Config.monitor_port))

    for line in sock.makefile():

        # check if this line is encrypted
        if line.startswith('1a'):
            line = karn.decrypt(line)
            if line is None: continue
            printer.directive(line, encrypted=True)
        else:
            printer.directive(line)

        # generate the response for this line
        response = generate_response(line)

        # if there is no response go get another line
        if response is None: continue

        printer.command(response + '\n', encrypted=karn)

        # if we have a valid karn we can encrypt the response
        if karn:
            response = karn.encrypt(response)

        # add the newline and send the response
        response += '\n'
        sock.send(response)

        if exit: break

    sock.close()
```

Listing 1: Client

```
import argparse
import sys
```

```python
import socket
import SocketServer
import platform
import subprocess
from config import Config
from config import checksums
from printer import Printer
from diffie_hellman import DHE
from karn import Karn
from fiat import Verifier
import parse_log

class tcp_handler(SocketServer.StreamRequestHandler):

    def setup(self):

        self.dhe = DHE()
        self.karn = None
        self.verifier = Verifier()

        self.printer = Printer("server")

        self.exit = False

        SocketServer.StreamRequestHandler.setup(self)

    def finish(self):

        SocketServer.StreamRequestHandler.finish(self)

    def generate_response(self, line):

        line = line.strip()
        directive, args = [x.strip() for x in line.split(':', 1)]

        if directive == "REQUIRE":
            if args == "IDENT":
                return "IDENT %s %s" % (Config.ident, self.dhe.public_key)
            elif args == "ALIVE":
                return "ALIVE %s" % Config.cookie
            elif args == "ROUNDS":
                return "ROUNDS %s" % Config.num_rounds
            elif args == "SUBSET_A":
                return "SUBSET_A %s" % ' '.join(str(x) for x in self.verifier.subset_a)
            elif args == "TRANSFER_RESPONSE":
                if self.verifier.good():
                    return "TRANSFER_RESPONSE ACCEPT"
                else:
                    return "TRANSFER_RESPONSE DECLINE"
            elif args == "QUIT":
                return "QUIT"
            else:
                self.printer.error("Unknown require: " + line)
        elif directive == "RESULT":
            args = args.split(' ', 1)
            if args[0] == "IDENT":
                self.dhe.monitor_key(args[1])
                self.karn = Karn(self.dhe.secret)
                self.printer.info("Setup secret key")
            elif args[0] == "ALIVE" and args[1] == "Identity has been verified.":
                self.printer.info("Alive verified")
            elif args[0] == "QUIT":
                self.printer.info("server has quit")
            elif args[0] == "SUBSET_K":
                self.verifier.subset_k = [int(x) for x in args[1].split()]
            elif args[0] == "SUBSET_J":
                self.verifier.subset_j = [int(x) for x in args[1].split()]
            elif args[0] == "PUBLIC_KEY":
                self.verifier.v = int(args[1].split()[0], 32)
                self.verifier.n = int(args[1].split()[1], 32)
            elif args[0] == "AUTHORIZE_SET":
                self.verifier.authorize_set = [int(x) for x in args[1].split()]
            else:
                self.printer.error("Unknown result")
        elif directive == "PARTICIPANT_PASSWORD_CHECKSUM":
            self.printer.info("Got checksum: %s" % args)
            if args not in checksums:
                self.printer.error("INVALID CHECKSUM")
                self.exit = True
                return ""
        elif directive == "WAITING":
```

```
                pass
            elif directive == "COMMENT":
                pass
            else:
                self.printer.error("Unknown directive: " + line)

            return None

    def handle(self):

        for line in self.rfile:

            # check if this line is encrypted
            if line.startswith('1a'):
                line = self.karn.decrypt(line)
                if line is None: continue
                self.printer.directive(line, encrypted=True)
            else:
                self.printer.directive(line)

            # generate the response for this line
            response = self.generate_response(line)

            # if there is no response go get another line
            if response is None: continue

            self.printer.command(response + '\n', encrypted=self.karn)

            # if we have a valid karn we can encrypt the response
            if self.karn:
                response = self.karn.encrypt(response)

            # add the newline and send the response
            response += '\n'
            self.wfile.write(response)

            if self.exit:
                self.printer.error("Exiting!")
                break

if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument('--ident', default="mtest16")
    parser.add_argument('--logfile', default="/home/httpd/html/final.log.8180")
    args = parser.parse_args()

    if args.ident not in Config.accounts:
        print "Invalid ident"
        sys.exit(1)

    Config.ident       = Config.accounts[args.ident].ident
    Config.server_port = Config.accounts[args.ident].port

    parse_log.parse(args.logfile)
    Config.cookie = parse_log.cookies[args.ident.lower()]
    Config.password = parse_log.passes[args.ident.lower()]
    print "Using cookie: '%s'" % Config.cookie

    print "Starting server on %s:%s" % (Config.server_ip, Config.server_port)
    server = SocketServer.ThreadingTCPServer((Config.server_ip, Config.server_port), tcp_handler)
    server.serve_forever()
```

Listing 2: Server

```
from random import getrandbits
from base32 import base32

class DHE:

    def __init__(self, key=None):

        self.p = 0x96C99B60C4F823707B47A848472345230C5B25103DC37412A701833E8FF5C567A53A41D0B37B10F0060D50F4131C57CF1FD
        self.g = 0x2C900DF142E2B839E521725585A92DC0C45D6702A48004A917F74B73DB26391F20AEAE4C6797DD5ABFF0BFCAECB29554248

        """
        Generate a new private key if it wasn't given
        """
        if (key):
```

```
            self.private_key = int(key, 32)
        else:
            self.private_key = getrandbits(512)

        """
        Public Key = g**x % p
        """
        self.public_key = base32(pow(self.g, self.private_key, self.p))

    def monitor_key(self, key):
        """
        Takes in the monitors public key in Base 32
        """
        self.secret = pow(int(key,32), self.private_key, self.p)
```

Listing 3: Diffie Hellman

```
from hashlib import sha1
from itertools import izip_longest
import struct
import base64
import string
from printer import Printer

BLOCK_SIZE = 40
GUARD_BYTE = 42

class Karn:

    def __init__(self, key):

        # Convert the integer key into a hex string
        # If the length of it is odd we need to left pad a '0'
        key_hex = '%x' % key
        if len(key_hex) & 1:
            key_hex = '0' + key_hex

        # Convert the hex key string into byte array
        key = bytearray.fromhex(key_hex)

        # java BigInteger.toByteArray() is signed so it pads if MSB is 1
        # http://stackoverflow.com/a/8544521/253650
        if key[0] & (1<<7):
            key = bytearray([0]) + key

        # Split key into two halfs
        # Monitor drops last byte if len is odd (monitor/Cipher.java:87)
        self.key_left  = key[:len(key)/2]
        self.key_right = key[len(key)/2:(len(key)/2)*2]

        self.printer = Printer("karn")

    def encrypt(self, message):

        # Start the output with the guard byte
        output = bytearray([GUARD_BYTE])

        # Break message into blocks and process each one
        for block in self._grouper(message, BLOCK_SIZE, '\0'):

            # Convert the block into an array of bytes
            block = bytearray(block)

            # Divide block into left and right half
            block_left  = block[:BLOCK_SIZE/2]
            block_right = block[BLOCK_SIZE/2:]

            # Hash the left plaintext plus the left key
            digest = sha1(block_left + self.key_left).digest()

            # XOR the digest with the right plaintext
            cipher_right = bytearray([ord(d)^b for d,b in zip(digest, block_right)])

            # Hash the right cipher plus the right key
            digest = sha1(cipher_right + self.key_right).digest()

            # XOR the digest with the left plaintext
            cipher_left = bytearray([ord(d)^b for d,b in zip(digest, block_left)])

```

```
            output += cipher_left
            output += cipher_right

        # Convert output to a hex string
        output = '0x'+''.join('%02X' % x for x in output)

        # Convert the hex string to an integer and then conver to base 32
        return self._baseN(int(output, 16), 32)

    def decrypt(self, message):

        # Convert the message from base 32 to hex
        message = '%x' % int(message, 32)
        if len(message) & 1:
            message = '0' + message

        # Convert the hex string to a byte array
        message = bytearray.fromhex(message)

        # If the first byte isn't the guard byte we are done
        if message[0] != GUARD_BYTE:
            self.printer.error("Did not find guard_byte!")
            return None

        # Remove the guard byte
        message = message[1:]

        output = bytearray()

        # Break message into blocks and process each one
        for block in self._grouper(message, BLOCK_SIZE, '\0'):

            # Convert the block into an array of bytes
            block = bytearray(block)

            # Divide block into left and right half
            block_left  = block[:BLOCK_SIZE/2]
            block_right = block[BLOCK_SIZE/2:]

            # Find digest of cipher right and key right
            digest = sha1(block_right + self.key_right).digest()

            # XOR the digest with cipher left
            text_left = bytearray([ord(d)^b for d,b in zip(digest, block_left)])

            # Find the digest of text left and key left
            digest = sha1(text_left + self.key_left).digest()

            # XOR the digest with cipher right
            text_right = bytearray([ord(d)^b for d,b in zip(digest, block_right)])

            output += text_left
            output += text_right

        # Remove the padding
        output = str(output).split('\0')[0]

        # Make sure the plaintext is normal printable text
        if not all(x in string.printable for x in output):
            self.printer.error("Unable to decrypt: %r" % output)
            return None

        return output

    def _grouper(self, iterable, n, fillvalue=None):
        """
        Collect data into fixed-length chunks or blocks
        grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx
        http://docs.python.org/2/library/itertools.html#recipes
        """
        args = [iter(iterable)] * n
        return izip_longest(fillvalue=fillvalue, *args)

    def _baseN(self, num,b,numerals="0123456789abcdefghijklmnopqrstuvwxyz"):
        """
        Return 'num' in base 'b'
        http://stackoverflow.com/a/2267428
        """
        return ((num == 0) and numerals[0]) or (self._baseN(num // b, b, numerals).lstrip(numerals[0]) + numerals[num

if __name__ == "__main__":
```

```
    text = "this is a test"
    key = 123456789
    enc =  "1avfbcuej96oo1m2vc04rnlin6rpurc2pu7ac8h42dhr8l13ahdfcbev3a5sj74o85"

    k = Karn(key)

    print "--- Testing Correct ---"
    e = k.encrypt(text)
    assert e == enc
    d = k.decrypt(e)
    assert d == text

    print "--- Testing No Guard Byte ---"
    e = '0' + enc[1:]
    d = k.decrypt(e)
    assert d == None

    print "--- Testing Corrupted Message ---"
    e = enc[:10] + "00000" + enc[15:]
    d = k.decrypt(e)
    assert d == None

    print "--- Done ---"
```

Listing 4: Karn

```
import random
from config import Config

class Prover:

    '''
    http://pages.swcp.com/~mccurley/talks/msri2/node24.html
    http://friedo.szm.com/krypto/AC/ch21/21-01.html
    '''

    p = 18446744073709551253
    q = 18446744073709551557
    n = p * q

    def __init__(self):

        self.s = random.randrange(2<<64)
        self.v = pow(self.s, 2, self.n)
        self.subset_a = []
        self.r_set = []
        self.rounds = 10

    def authorize_set(self):
        # generate the random set
        self.r_set = [random.randrange(2<<64) for _ in range(self.rounds)]
        # calculate the authorize set
        auth_set = [pow(r, 2, self.n) for r in self.r_set]
        # return it as a string
        return ' '.join(str(x) for x in auth_set)

    def subset_k(self):
        k = [(self.s * self.r_set[i]) % self.n for i in self.subset_a]
        return ' '.join(str(x) for x in k)

    def subset_j(self):
        j = [self.r_set[i] % self.n for i in set(range(self.rounds)) - set(self.subset_a)]
        return ' '.join(str(x) for x in j)


class Verifier:

    def __init__(self):

        self.rounds = Config.num_rounds
        self.v = 0
        self.n = 0

        self.authorize_set = []
        self.subset_j = []
        self.subset_k = []

        self.subset_a = sorted(random.sample(range(self.rounds), self.rounds // 2))
```

```python

    def good(self):

        # check that the public key is our own
        if self.n != Prover.n:
            print "public keys dont match!"
            return False

        j = iter(self.subset_j)
        k = iter(self.subset_k)
        for i in range(self.rounds):
            if i in self.subset_a:
                if pow(next(k), 2, self.n) != (self.v * self.authorize_set[i]) % self.n:
                    return False
            elif pow(next(j), 2, self.n) != self.authorize_set[i]:
                return False

        return True
```

Listing 5: Fiat-Shamir - Zero Knowledge Proof

```python
from config import Config

class Printer:

    BOLD    = '\033[1m'
    BLACK   = '\033[30m'
    RED     = '\033[31m'
    GREEN   = '\033[32m'
    YELLOW  = '\033[33m'
    BLUE    = '\033[34m'
    PURPLE  = '\033[35m'
    CYAN    = '\033[36m'
    WHITE   = '\033[37m'
    DEFAULT = '\033[0m'

    def __init__(self, who=""):
        self.who = who

    def directive(self, string, encrypted=False):
        if encrypted:
            label = "<E<"
        else:
            label = "<<<"
        self._print(string, label, self.GREEN)

    def command(self, string, encrypted=False):
        if encrypted:
            label = ">E>"
        else:
            label = ">>>"
        self._print(string, label, self.BLUE)

    def error(self, string):
        string = string.strip()
        self._print(string, "---", self.RED)

    def info(self, string):
        string = string.strip()
        self._print(string, "---", self.YELLOW)

    def _print(self, string, label, color):
        string = string.strip()
        if string == "": return
        print "[" + Config.ident + "] " + label + "  " + color + string + self.DEFAULT
```

Listing 6: Printer

```python
def base32(num):
    return _baseN(num, 32)

def _baseN(num, b, numerals="0123456789abcdefghijklmnopqrstuvwxyz"):
    """
    Return 'num' in base 'b'
    http://stackoverflow.com/a/2267428
    """
    return ((num == 0) and numerals[0]) or (_baseN(num // b, b, numerals).lstrip(numerals[0]) + numerals[num % b])
```

Listing 7: Base 32

```
import hashlib
import socket

checksums = []

class Account:
    def __init__(self, ident, password, cookie, port):
        global checksums
        self.ident = ident
        self.password = password
        self.cookie = cookie
        self.port = port
        checksums.append(hashlib.sha1(password).hexdigest())

class Config:
    monitor_dns = "gauss.ececs.uc.edu"
    monitor_ip = socket.gethostbyname(monitor_dns)
    monitor_port = 8180
    server_ip = socket.gethostbyname(socket.getfqdn())
    server_port = 20167
    accounts = {
            "EDSNOWDEN":     Account("EDSNOWDEN", "8Y6]%D[:T%2!.P^$6AGDEHMCZICN;OOX-]UOJU=^G-", "ATPL5QCPINFHAWYQWXN",
            "GROUND_WATER":  Account("GROUND_WATER", "Y9JUY8A`L?+;RK9O$;SWRLO_M'I-[9MP)IK=*]4-E!", "QNU0DWVPNQFGOSD1TC
            "CORNHOLIO":     Account("CORNHOLIO", "-S_:9J7R9X*/&7UDL'SX-'2?R=FOW2!T'I8.Q7F?PA", "W6M2D5OCZAJZWTOV3M8",
            "mtest16":       Account("mtest16", "12345", "C5K8GNM22KQ5XCFKVHF", 11121),
            "mtest17":       Account("mtest17", "12345", "J98Q82H1C458X6YAKAI", 11122),
            "mtest18":       Account("mtest18", "12345", "MGYKSTOKL4T106N0175", 11123),
    }

    ident = ""
    password = ""
    cookie = ""

    num_rounds = 5
```

Listing 8: Configuration File

```
import random
import gmpy2

'''
def is_prime(n):
    for k in range(50):
        a = random.randrange(1, n-1)
        a = gmpy2.powmod(a, n-1, n)

        if (a == 1):
            print "prime = %i\n" % (n)
            return True
        else:
            return False

def find_prime(lower_bound, upper_bound):
    for _ in range(1000000):
        if (~is_prime(lower_bound)):
            lower_bound = lower_bound + 1
            continue
        if (~is_prime(upper_bound)):
            upper_bound = upper_bound + 1
            continue
        n = random.randrange(lower_bound | 1, upper_bound, 2)
        if is_prime(n):
            print "%d | %d | %d\n" %(lower_bound, upper_bound, n)
            return

find_prime(2**32-1,2**64-1)
'''

def is_prime(n, k=50):
    '''Use Miller-Rabin Primality test to test if an integer is probably prime.

    The probability that this test will report a prime number as composite is 4**-k'''
    if n == 2 or n == 3:
```

```
        return True
    if n < 2:
        return False

    #divide by some small primes first to fast track most composite numbers
    for i in (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 ,47):
        if n % i == 0:
            return False

    s = 1
    d = n - 1

    while d % 2 == 0:
        s += 1
        d = d >> 1

    for i in xrange(k):
        x = pow(random.randint(2,n-2), d, n)
        if x != 1 and x != n - 1:
            for _ in xrange(1, s):
                x = pow(x, 2, n)
                if x == 1:
                    return False
                if x == n - 1:
                    break
            else:
                return False
    return True

def find_prime(lower_bound, upper_bound):
    for _ in range(1000):
        n = random.randrange(lower_bound | 1, upper_bound, 2)
        print "%d | %d | %d\n" %(lower_bound, upper_bound, n)
        if is_prime(n):
            return n
    raise FiatShamirError('Could not find a prime number.')

find_prime(2**64-363,2**64-59)
```

Listing 9: Test 1

```
from random import randrange

class fiat:
    def __init__(self):
        '''
        http://pages.swcp.com/~mccurley/talks/msri2/node24.html
        http://friedo.szm.com/krypto/AC/ch21/21-01.html
        '''
        p = 18446744073709551253
        q = 18446744073709551557
        n = p * q
        self.s = randrange(self.n)
        self.v = pow(self.s, 2, self.n)
        self.rounds = 50

    # client
    def prover(self):

        self.x = 0
        self.y = 0
        self.r = 0
        self.z_n = ()
        self.z_n = tuple(randrange(self.n) for _ in xrange(self.rounds))

        # A: picks random r in Z_n *, sends x=r^2 mod n to B
        def public_authen(self):
            for self.r in self.z_n:
                self.x = pow(r, 2, self.n)
                yield self.x

        #a A sends y to B, where If c=0, y=r, else y=rs mod n
        def send(self):
            if (~self.c):
                self.y = self.r
                yield self.y
            else:
                self.y = (self.r*self.s) % self.n
                yield self.y
```

```

    # server
    def verifier(self):

        self.c = 0
        self.autorization = ()
        self.key_i = ()
        self.key_j = ()

        # B checks x!=0 and sends random c in {0,1} to A
        def check(self):
            if (self.x):
                self.c = randrange(0,1)
                yield self.c

        # B accept if y^2 congruent xv^c mod n
        def accept(self):
            if (self.n == fiat.n): return True

            if (pow(self.y,2,self.n) == (self.x*pow(self.v,self.c))):
                return True

            i = iter(self.key_i)
            j = iter(self.key_j)
            for r in xrange(self.rounds):
                if pow(next(i), 2, self.n) == ((self.v * self.autorization[i]) % self.n):
                    return True
                elif pow(next(j), 2, self.n) == (self.autorization[i]):
                    return True
                else:
                    return False
```

Listing 10: Test 2

References

[1] http://www.cs.princeton.edu/courses/archive/fall07/cos433/lec15.pdf