CS6053 - Network Security

Stack Attack

April 20, 2014

Contents

List of Figures

List of Tables

1 Maintaining Security

In order to maintain a secure connection with the monitor we have implemented a few different algorithms: Diffie-Hellman, Karn Symmetric Cyrptosystem, and the Fiat-Shamir Algorithm.

1.1 Diffie-Hellman

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:

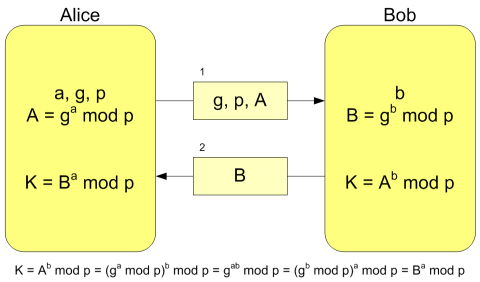

Figure 1: Diffie-Hellman Key Exchange

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:

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

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.

1.2 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 block 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.

One interesting problem we came across is that the monitor is using a Java BigInteger type to store its key and the BigInteger is signed so Java adds a pad byte to maintain its signedness. This results in a key that is one byte longer than we expect. Since we are using Python we have to manually add the extra pad byte. This should really be fixed in the monitor (`monitor/Cipher.java:87`) so that the pad byte is ignored.

1.3 Zero-Knowledge Proof

A zero-knowledge proof is a proof that 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:

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

We initially had a hard time implementing this and ended up closely following AJ Alts and Ryan Childs implementation [2] from a few years ago.

2 Attacking Accounts

We entered this competition from a purely defensive standpoint, mostly due to the fact that we didn't have our code fully working until a few days before the competition. When the competition began we waited for a while to see what the activity of the other players would be. After seeing basically not activity for almost an hour we decided to just start our script that would transfer funds between our accounts. As other players connected we manually recorded their cookies (later wrote a script to harvest them) in anticipation of being able to use them later. It wasn't long before somehow our password was changed. At the time this was really baffling because we had never sent our password to the monitor and you need the old password to create a new one. As we later found out the team that got root on gauss had full control of the monitor. At this point we were unable to do much. We added the functionality to parse the logs for our latest cookie and password in an attempt to recover our accounts. This was fruitless, however, being that every time the other team logged in with our account they immediately changed the password again causing the cookie to change. We left our script running anyway in the hopes that maybe their strategy would change and we would be able to grab the new password to our account. Needless to say we were never able to recover our account and the points we had accrued early in the competition were taken back by the monitor over time as our server was not able to respond to the alive requests.

Figure 2: Screenshot of script transferring points between our three accounts

3 Appendix

3.1 Python Code

```
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
```

```
76              return rt
77          if manual_mode and authenticated:
78              return raw_input('Command: ')
79      elif directive == "COMMENT":
80          pass
81      elif directive == "COMMAND_ERROR":
82          exit = True
83          printer.error(line)
84          return ""
85      else:
86          exit = True
87          printer.error("Unknown directive: " + line)
88          return ""
89
90      return None
91
92  if __name__ == "__main__":
93
94      parser = argparse.ArgumentParser()
95      parser.add_argument('--ident', default="mtest16")
96      parser.add_argument('--transfer', nargs=3, metavar=('TO', 'AMOUNT', 'FROM'))
97      parser.add_argument('--manual', action='store_true')
98      parser.add_argument('--alive', action='store_true')
99      parser.add_argument('--logfile', default="/home/httpd/html/final.log.8180")
100     args = parser.parse_args()
101
102     manual_mode = args.manual
103     transfer = args.transfer
104     only_do_alive = args.alive
105
106     if args.ident not in Config.accounts:
107         print "Invalid ident"
108         sys.exit(1)
109
110     Config.ident = args.ident
111     Config.server_port = Config.accounts[args.ident].port
112
113     parse_log.parse(args.logfile)
114     Config.cookie = parse_log.cookies[args.ident.lower()]
115     Config.password = parse_log.passes[args.ident.lower()]
116
117     print "Using cookie: '%s'" % Config.cookie
118
119     sock = socket.create_connection((Config.monitor_ip, Config.monitor_port))
120
121     for line in sock.makefile():
122
123         # check if this line is encrypted
124         if line.startswith('1a'):
125             line = karn.decrypt(line)
126             if line is None: continue
127             printer.directive(line, encrypted=True)
128         else:
129             printer.directive(line)
130
131         # generate the response for this line
132         response = generate_response(line)
133
134         # if there is no response go get another line
135         if response is None: continue
136
137         printer.command(response + '\n', encrypted=karn)
138
139         # if we have a valid karn we can encrypt the response
140         if karn:
141             response = karn.encrypt(response)
142
143         # add the newline and send the response
144         response += '\n'
145         sock.send(response)
146
147         if exit: break
148
149     sock.close()
```

Listing 1: Client

```
1  import argparse
2  import sys
```

```
3  import socket
4  import SocketServer
5  import platform
6  import subprocess
7  from config import Config
8  from config import checksums
9  from printer import Printer
10 from diffie_hellman import DHE
11 from karn import Karn
12 from fiat import Verifier
13 import parse_log
14
15 class tcp_handler(SocketServer.StreamRequestHandler):
16
17     def setup(self):
18
19         self.dhe = DHE()
20         self.karn = None
21         self.verifier = Verifier()
22
23         self.printer = Printer("server")
24
25         self.exit = False
26
27         SocketServer.StreamRequestHandler.setup(self)
28
29     def finish(self):
30
31         SocketServer.StreamRequestHandler.finish(self)
32
33     def generate_response(self, line):
34
35         line = line.strip()
36         directive, args = [x.strip() for x in line.split(':', 1)]
37
38         if directive == "REQUIRE":
39             if args == "IDENT":
40                 return "IDENT %s %s" % (Config.ident, self.dhe.public_key)
41             elif args == "ALIVE":
42                 return "ALIVE %s" % Config.cookie
43             elif args == "ROUNDS":
44                 return "ROUNDS %s" % Config.num_rounds
45             elif args == "SUBSET_A":
46                 return "SUBSET_A %s" % ' '.join(str(x) for x in self.verifier.subset_a)
47             elif args == "TRANSFER_RESPONSE":
48                 if self.verifier.good():
49                     return "TRANSFER_RESPONSE ACCEPT"
50                 else:
51                     return "TRANSFER_RESPONSE DECLINE"
52             elif args == "QUIT":
53                 return "QUIT"
54             else:
55                 self.printer.error("Unknown require: " + line)
56         elif directive == "RESULT":
57             args = args.split(' ', 1)
58             if args[0] == "IDENT":
59                 self.dhe.monitor_key(args[1])
60                 self.karn = Karn(self.dhe.secret)
61                 self.printer.info("Setup secret key")
62             elif args[0] == "ALIVE" and args[1] == "Identity has been verified.":
63                 self.printer.info("Alive verified")
64             elif args[0] == "QUIT":
65                 self.printer.info("server has quit")
66             elif args[0] == "SUBSET_K":
67                 self.verifier.subset_k = [int(x) for x in args[1].split()]
68             elif args[0] == "SUBSET_J":
69                 self.verifier.subset_j = [int(x) for x in args[1].split()]
70             elif args[0] == "PUBLIC_KEY":
71                 self.verifier.v = int(args[1].split()[0], 32)
72                 self.verifier.n = int(args[1].split()[1], 32)
73             elif args[0] == "AUTHORIZE_SET":
74                 self.verifier.authorize_set = [int(x) for x in args[1].split()]
75             else:
76                 self.printer.error("Unknown result")
77         elif directive == "PARTICIPANT_PASSWORD_CHECKSUM":
78             self.printer.info("Got checksum: %s" % args)
79             if args not in checksums:
80                 self.printer.error("INVALID CHECKSUM")
81                 self.exit = True
82                 return ""
83         elif directive == "WAITING":
```

```
84              pass
85          elif directive == "COMMENT":
86              pass
87          else:
88              self.printer.error("Unknown directive: " + line)
89
90          return None
91
92      def handle(self):
93
94          for line in self.rfile:
95
96              # check if this line is encrypted
97              if line.startswith('1a'):
98                  line = self.karn.decrypt(line)
99                  if line is None: continue
100                 self.printer.directive(line, encrypted=True)
101             else:
102                 self.printer.directive(line)
103
104             # generate the response for this line
105             response = self.generate_response(line)
106
107             # if there is no response go get another line
108             if response is None: continue
109
110             self.printer.command(response + '\n', encrypted=self.karn)
111
112             # if we have a valid karn we can encrypt the response
113             if self.karn:
114                 response = self.karn.encrypt(response)
115
116             # add the newline and send the response
117             response += '\n'
118             self.wfile.write(response)
119
120             if self.exit:
121                 self.printer.error("Exiting!")
122                 break
123
124 if __name__ == "__main__":
125
126     parser = argparse.ArgumentParser()
127     parser.add_argument('--ident', default="mtest16")
128     parser.add_argument('--logfile', default="/home/httpd/html/final.log.8180")
129     args = parser.parse_args()
130
131     if args.ident not in Config.accounts:
132         print "Invalid ident"
133         sys.exit(1)
134
135     Config.ident       = Config.accounts[args.ident].ident
136     Config.server_port = Config.accounts[args.ident].port
137
138     parse_log.parse(args.logfile)
139     Config.cookie = parse_log.cookies[args.ident.lower()]
140     Config.password = parse_log.passes[args.ident.lower()]
141     print "Using cookie: '%s'" % Config.cookie
142
143     print "Starting server on %s:%s" % (Config.server_ip, Config.server_port)
144     server = SocketServer.ThreadingTCPServer((Config.server_ip, Config.server_port), tcp_handler)
145     server.serve_forever()
```

Listing 2: Server

```
1   """
2   Diffie Hellman Key Exchange
3   """
4
5   from random import getrandbits
6   from base32 import base32
7
8   class DHE:
9
10      def __init__(self, key=None):
11
12          self.p = 0x96C99B60C4F823707B47A848472345230C5B25103DC37412A701833E8FF5C567A53A41D0B37B10F0060D50F4131C57CF1FD
13          self.g = 0x2C900DF142E2B839E521725585A92DC0C45D6702A48004A917F74B73DB26391F20AEAE4C6797DD5ABFF0BFCAECB29554248
14
```

```
        """
        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

```
"""
Karn Symmetric Key Cryptosystem
"""

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
```

```
59              cipher_right = bytearray([ord(d)^b for d,b in zip(digest, block_right)])
60
61              # Hash the right cipher plus the right key
62              digest = sha1(cipher_right + self.key_right).digest()
63
64              # XOR the digest with the left plaintext
65              cipher_left = bytearray([ord(d)^b for d,b in zip(digest, block_left)])
66
67              output += cipher_left
68              output += cipher_right
69
70          # Convert output to a hex string
71          output = '0x'+''.join('%02X' % x for x in output)
72
73          # Convert the hex string to an integer and then conver to base 32
74          return self._baseN(int(output, 16), 32)
75
76      def decrypt(self, message):
77
78          # Convert the message from base 32 to hex
79          message = '%x' % int(message, 32)
80          if len(message) & 1:
81              message = '0' + message
82
83          # Convert the hex string to a byte array
84          message = bytearray.fromhex(message)
85
86          # If the first byte isn't the guard byte we are done
87          if message[0] != GUARD_BYTE:
88              self.printer.error("Did not find guard_byte!")
89              return None
90
91          # Remove the guard byte
92          message = message[1:]
93
94          output = bytearray()
95
96          # Break message into blocks and process each one
97          for block in self._grouper(message, BLOCK_SIZE, '\0'):
98
99              # Convert the block into an array of bytes
100             block = bytearray(block)
101
102             # Divide block into left and right half
103             block_left  = block[:BLOCK_SIZE/2]
104             block_right = block[BLOCK_SIZE/2:]
105
106             # Find digest of cipher right and key right
107             digest = sha1(block_right + self.key_right).digest()
108
109             # XOR the digest with cipher left
110             text_left = bytearray([ord(d)^b for d,b in zip(digest, block_left)])
111
112             # Find the digest of text left and key left
113             digest = sha1(text_left + self.key_left).digest()
114
115             # XOR the digest with cipher right
116             text_right = bytearray([ord(d)^b for d,b in zip(digest, block_right)])
117
118             output += text_left
119             output += text_right
120
121         # Remove the padding
122         output = str(output).split('\0')[0]
123
124         # Make sure the plaintext is normal printable text
125         if not all(x in string.printable for x in output):
126             self.printer.error("Unable to decrypt: %r" % output)
127             return None
128
129         return output
130
131     def _grouper(self, iterable, n, fillvalue=None):
132         """
133         Collect data into fixed-length chunks or blocks
134         grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx
135         http://docs.python.org/2/library/itertools.html#recipes
136         """
137         args = [iter(iterable)] * n
138         return izip_longest(fillvalue=fillvalue, *args)
139
```

```
140     def _baseN(self, num,b,numerals="0123456789abcdefghijklmnopqrstuvwxyz"):
141         """
142         Return 'num' in base 'b'
143         http://stackoverflow.com/a/2267428
144         """
145         return ((num == 0) and numerals[0]) or (self._baseN(num // b, b, numerals).lstrip(numerals[0]) + numerals[num
146
147 if __name__ == "__main__":
148
149     text = "this is a test"
150     key = 123456789
151     enc =  "1avfbcuej96oo1m2vc04rnlin6rpurc2pu7ac8h42dhr8l13ahdfcbev3a5sj74o85"
152
153     k = Karn(key)
154
155     print "--- Testing Correct ---"
156     e = k.encrypt(text)
157     assert e == enc
158     d = k.decrypt(e)
159     assert d == text
160
161     print "--- Testing No Guard Byte ---"
162     e = '0' + enc[1:]
163     d = k.decrypt(e)
164     assert d == None
165
166     print "--- Testing Corrupted Message ---"
167     e = enc[:10] + "00000" + enc[15:]
168     d = k.decrypt(e)
169     assert d == None
170
171     print "--- Done ---"
```

Listing 4: Karn

```
1   """
2   Fiat-Shamir Zero Knowledge Authentication
3   """
4
5   import random
6   from config import Config
7
8   class Prover:
9
10      '''
11      http://pages.swcp.com/~mccurley/talks/msri2/node24.html
12      http://friedo.szm.com/krypto/AC/ch21/21-01.html
13      '''
14
15      p = 18446744073709551253
16      q = 18446744073709551557
17      n = p * q
18
19      def __init__(self):
20
21          self.s = random.randrange(2<<64)
22          self.v = pow(self.s, 2, self.n)
23          self.subset_a = []
24          self.r_set = []
25          self.rounds = 10
26
27      def authorize_set(self):
28          # generate the random set
29          self.r_set = [random.randrange(2<<64) for _ in range(self.rounds)]
30          # calculate the authorize set
31          auth_set = [pow(r, 2, self.n) for r in self.r_set]
32          # return it as a string
33          return ' '.join(str(x) for x in auth_set)
34
35      def subset_k(self):
36          k = [(self.s * self.r_set[i]) % self.n for i in self.subset_a]
37          return ' '.join(str(x) for x in k)
38
39      def subset_j(self):
40          j = [self.r_set[i] % self.n for i in set(range(self.rounds)) - set(self.subset_a)]
41          return ' '.join(str(x) for x in j)
42
43
44  class Verifier:
```

```
45
46      def __init__(self):
47
48          self.rounds = Config.num_rounds
49          self.v = 0
50          self.n = 0
51
52          self.authorize_set = []
53          self.subset_j = []
54          self.subset_k = []
55
56          self.subset_a = sorted(random.sample(range(self.rounds), self.rounds // 2))
57
58      def good(self):
59
60          # check that the public key is our own
61          if self.n != Prover.n:
62              print "public keys dont match!"
63              return False
64
65          j = iter(self.subset_j)
66          k = iter(self.subset_k)
67          for i in range(self.rounds):
68              if i in self.subset_a:
69                  if pow(next(k), 2, self.n) != (self.v * self.authorize_set[i]) % self.n:
70                      return False
71              elif pow(next(j), 2, self.n) != self.authorize_set[i]:
72                  return False
73
74          return True
```

Listing 5: Fiat-Shamir - Zero Knowledge Proof

```
1   """
2   Prints colored and properly formatted messages to stdout
3   """
4
5   from config import Config
6
7   class Printer:
8
9       BOLD    = '\033[1m'
10      BLACK   = '\033[30m'
11      RED     = '\033[31m'
12      GREEN   = '\033[32m'
13      YELLOW  = '\033[33m'
14      BLUE    = '\033[34m'
15      PURPLE  = '\033[35m'
16      CYAN    = '\033[36m'
17      WHITE   = '\033[37m'
18      DEFAULT = '\033[0m'
19
20      def __init__(self, who=""):
21          self.who = who
22
23      def directive(self, string, encrypted=False):
24          if encrypted:
25              label = "<E<"
26          else:
27              label = "<<<"
28          self._print(string, label, self.GREEN)
29
30      def command(self, string, encrypted=False):
31          if encrypted:
32              label = ">E>"
33          else:
34              label = ">>>"
35          self._print(string, label, self.BLUE)
36
37      def error(self, string):
38          string = string.strip()
39          self._print(string, "---", self.RED)
40
41      def info(self, string):
42          string = string.strip()
43          self._print(string, "---", self.YELLOW)
44
45      def _print(self, string, label, color):
46          string = string.strip()
```

```
47          if string == "": return
48          print "[" + Config.ident + "] " + label + "  " + color + string + self.DEFAULT
```

Listing 6: Printer

```
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

References

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

[2] https://github.com/ajalt/PWNmlete-2011/blob/master/fiat_shamir.py