A Firewall Access Definition Language

Jianyu You
NEC Systems Laboratory, Irving, Texas, USA
Ernst L. Leiss
Department of Computer Science, University of Houston, Houston, Texas, USA

Abstract

Securing resources against unauthorized access and/or use is a major concern of every organization that uses computer networks. To protect internal networks from external attacks, firewalls are utilized since they restrict network access while letting legitimate users have unencumbered access. This paper describes the design of a language to specify firewall access control lists and alarms. A procedure to check consistency among access permissions is also developed.

1. Introduction

Firewalls are hardware/software systems that allow one to isolate a subnet from a larger network in a way that restricts access to certain groups of users. Securing resources against unauthorized access and/or use is a major concern of every organization that uses computer networks. To protect internal networks from external attacks, firewalls are utilized since they restrict network access while letting legitimate users have unencumbered access. Firewalls can also be used to log security auditing information about connections and operations. For more details about firewalls, their uses, restrictions and limitations, as well as related issues, see [5].

Currently, many firewalls use a list format to define the access control list; this is quite inconvenient and can often lead to confusion. Also, when fields are numerous (including source and destination's IP addresses, masks, protocols, port numbers, users, access time, authentication methods, etc.), it is error prone and inconvenient to define the access control list. For example, SOCKS v4 [6] defines the following format for access control:

action [*=userlist] src_addr src_mask [dst_addr dst_mask] [op dst_port] [: shell_cmd]

Thus

permit *=boss,root 1.2.3.4 255.255.255.255 11.12.13.14 255.255.255.255 eq telnet

means that the SOCKS server allows requests from a local user whose effective id is either boss or root, the source IP address must be 1.2.3.4 exactly, the destination IP address must be 11.12.13.14 exactly, and the service requested must be telnet.

The Firewall Access Definition Language (FADL) defined in this paper aims at providing sequential statements familiar to system administrators to generate the access control list. Though the grammar is designed to be simple at the beginning, the language construct will allow expansion to meet future requirements.

The advantages of using a language for defining a firewall's access control list include the following:

- It offers a clearer description of the access through the use of natural IF..THEN..ELSE statements.
- It is easier to use; for example, application names can be used instead of specifying protocol and port numbers.
- It is flexible: if an additional variable has to be taken into consideration, we can define a global variable instead of adding a field to every line in the conventional access control list specification.

In our FADL, the configuration file of a firewall under consideration has two sections:

(1) Connection permission
(2) Alarm specification

The FADL consists of only three language constructs to define these sections: the assignment statement, the conditional statement, and the alarm specification.

Since the FADL is a little special purpose language, we can use criteria discussed in [1] and [2] for language design, namely, orthogonality, generality, parsimony, completeness, similarity, extensibility, and openness. These seven criteria are sketched below; they served as a guide to designing the FADL. These criteria refer to either the statements of the language or aspects of its implementation. For example, orthogonality refers to statements, and openness refers to statements and implementation.

In the context of programming languages, **orthogonality** means that different statements have different purposes, When designing a language, a single statement should be formulated for each function that the language is carry out.

The **generality** of a statement refers to the range of applications to which the statement applies. For example, a language may contain several statements to create program loops. There could be one for counting (such as the FOR-loop of BASIC), a WHILE-loop, a REPEAT-loop, and a DO...LOOP. The generality criterion would favor just one for all of the looping statements.

Another important characteristic of a programming language is **parsimony** which means that a language should not be wasteful; thus, no statement should duplicate functionality. A little language should not mimic languages with several statements to accomplish one task.

Completeness means that a language should meet all of the requirements of its specification to solve a problem.

Similarity means that the terminology used should reflect the domain of its application. The terminology and statements should be similar to those used in the problem domain of the language.

The designer and other users may have new ideas for, and enhancement suggestions to, a language in use. It is in our best interest to make it possible for the language to grow, to change, and to evolve. This is called **extensibility**.

The last criterion is that of **openness**, or the ability to share, the ability to communicate outside of one's boundaries like data transfers and external function calls.

In the next section, a description of our FADL is presented, then the grammar of the language is proposed. A method to check consistency among the statements is developed and the combination with alarm specifications using the FADL language construct is discussed.

2. Description of the FADL

Assignment statement

The variable naming is similar to that of `make`, `shell`, etc in Unix. The value of a variable is referenced by preceding its name by $. The syntax for an assignment is:

variable = value_list

The construct can be used to define lists, needed for our application. It is in a form familiar to system administrators; parsing codes from programs such as make are reusable. Two examples of the assignment statement are:

```
VAR1 = 123.445.566.323  123.344.566.666
SOURCE_LIST = .syl.dl.nec.com  .syl.sj.nec.com
              cnad.nj.nec.com \ .asl.dl.nec.com  123.455.678
```

In the future, the concatenation operation (such as in `VAR2 = $VAR1 $SOURCE_LIST`) and others may be supported. Predefined variables are used for an ACL's columns; they include:

`SOURCE`	the format is: `source address (SOURCE_MASK)`. If `SOURCE_MASK` is not specified, a default mask is computed.
`DESTINATION`	destination address. Similar to `SOURCE`.
`APPLICATION`	values can be `TELNET`, `FTP`, etc. or a triplet `(PROTOCOL, SOURCE_PORT, DESTINATION_PORT)`. Default values for services such as `TELNET`, `FTP` are defined.
`AUTH`	authentication method.
`USER`	user name.
`PERMIT`	has permission: YES or NO.
`ACCESS_TIME`	time in which access is allowed.
`ALERT_TIME`	time in which failed attempt numbers are kept.
`DENY[$SOURCE]`	number of denied requests during alert time by the source of the current attempt.
`DENY[$DESTINATION]`	number of denied requests during alert time by the destination of the current attempt.
`DENY[$USER]`	number of denied requests during alert time by the user of the current attempt.

These can be used to formulate statements.

Conditional statement

The conditional statement has the following form:

```
IF (condition)
THEN list-of-statements
ELSE list-of-statements
ENDIF
```

where list-of-statements is a sequence of assignment statements or other conditional statements (nested IF statements), and condition can be described by the following simplified grammar:

```
condition::= operand bin_operator operand
operand::= $variable | value_list
bin_operator::=  >= | <= | > | < | == | != | IN
```

The operator `IN` checks if the first operand matches some elements of the second operand (a list). Some examples are as follows:

```
IF ($SOURCE IN $SOURCE_LIST)
THEN PERMIT=YES
ELSE PERMIT=NO
ENDIF
IF ($SOURCE == 123.456.567.555)
THEN
     IF ($DESTINATION IN $VAR1)
     THEN
          APPLICATION=TELNET HTTP
          PERMIT=YES
     ENDIF
ELSE
     PERMIT=NO
ENDIF
```

Scope of variables

In order to isolate the effect of variables, the concept of block is defined; this is a set of statements enclosed by { }. Similar to C functions, variables used inside a block are local. Thus a user can define what is equivalent to a single access control line, without affecting variables in other blocks:

```
{SOURCE=123.144.145(SOURCE_MASK=255.255.255.0)
 DESTINATION=122.333.333.32
 PERMIT=YES
}
```

Variables defined outside a block are global to that block. If the authentication method is the same for the whole access control list, we can define a global authentication method and not bother to define it in every block. Variables defined as global can be used inside different blocks.

The need for checking consistency

When the number of lines in a usual access control list is small, say, 5 or 6 lines, the visual detection of inconsistency among them is easy. But as the number of lines grows above 20, visual inspection may be very difficult. Some possible errors such as those shown in the following program fragment may occur:

```
deny   source in .COM
...
allow source in .NEC.COM
```

The original goal was "deny everything from .COM but allow .NEC.COM". But here the `deny` line comes first. So `.NEC.COM` will be denied. The correct order should be

```
allow source in .NEC.COM
...
deny   source in .COM
```

The FADL compiler detects this situation using the unification concept taken from artificial intelligence. If the access control list consists of several lines, we check whether the second line is consistent with the first. Then we check that the third line is consistent with the first two lines, and so on. The worst-case time complexity is quadratic, but since it is done only once, during compiling time, it will not add computational burden later.

If we combine access control lists from two different firewalls, inconsistencies among them can be discovered by the same approach. A Host Access List can be defined based on the constructs discussed above. An example is the following:

```
AUTH=RevIp
{PERMIT=NO
 APPLICATION=HTTP
 SOURCE=.COM
 DESTINATION=101.101.101.102
}
SOURCE_LIST = 13.234.233.233 .syl.dl.nec.com .asl.dl.nec.com
{IF ($SOURCE IN $SOURCE_LIST)
 THEN
      IF ($DESTINATION == 23.222.233.233)
      THEN
           PERMIT=YES
           APLLICATION=(UDP,-,-)
      ELSE
           PERMIT=NO
      ENDIF
 ENDIF
}
```

```
{IF ($APPLICATION ==(TCP,-,23) &
    $DESTINATION == 29.444.444.555)
 THEN PERMIT=YES
 ELSE PERMIT=NO
 ENDIF
}
```

The resulting access control list is equivalent to the following in list format:

permit	protocol	auth	source	source_port	destination	dst_port
NO	TCP	RevIp	.COM	-	101.101.101.102	80
YES	UDP	RevIp	13.234.233.233	-	23.222.233.233	-
YES	UDP	RevIp	.syl.dl.nec.com	-	23.222.233.233	-
YES	UDP	RevIp	.asl.dl.nec.com	-	23.222.233.233	-
NO	-	RevIp	13.234.233.233	-	-	-
NO	-	RevIp	.syl.dl.nec.com	-	-	-
NO	-	RevIp	.asl.dl.nec.com	-	-	-
YES	TCP	RevIp	-	-	29.444.444.555	23
NO	-	RevIp	-	-	-	-

3. Grammar of the FADL

We give a formal specification of our firewall access definition language by formulating a context free the grammar .

```
acl_language --> block_list

block_list   --> assignment '\n' block_list | block '\n' block_list
                 | alarm  '\n' block_list | assignment | block | alarm |

block        --> '{' stmt_list '}'

stmt_list    --> assignment '\n' stmt_list  | conditional '\n' stmt_list
                  | assignment | conditional |

assignment   --> variable '=' atom_list

atom_list    --> atom atom_list |

conditional  --> 'IF' condition 'THEN' stmt_list 'ELSE' stmt_list 'ENDIF'
                 | 'IF' condition 'THEN' stmt_list 'ENDIF'

condition    --> condition '|' condition | condition '&' condition
                 | '~' condition | '(' condition ')' | predicate  | in_predicate

predicate    --> expression COMPARISON expression

comparison   --> '==' | '<=' |'>=' | '!=' | '<' | '>'

in_predicate --> expression 'IN' value | expression IN '(' atom_list ')'

expression   --> value | atom

alarm        --> 'WHEN' condition '{' action '}'

action_list  --> command '\n' action_list |

command      --> atom_list
```

lexical tokens:

```
atom        --> (character|digit|special)+

variable    --> character(character|digit|'_')*

value       --> $character(character|digit|'_')*

digit       --> '0' - '9'

character   --> 'a' - 'z' | 'A' - 'Z'

special     -->  '.' | ',' | '(' | ')' | '_' | ':'
```

4. Consistency Checking

One of the major purposes of the FADL is to check consistency among all the rules. To accomplish this task, predicate logic can be used (see, for example, [4]). In this section, the unification algorithm is described and a procedure to discover contradictions and redundancies in an ACL is developed. In predicate logic, we can represent real-world facts as statements written as "well-formed formulas". Consider the following set of sentences:

(1) User John was denied by host H.

(2) All users should be allowed to access H

In predicate logic, sentence (1) can be represented by

deny(John,H)

deny(John,H) captures the critical fact of the sentence but fails to capture some of the information in the English sentence, namely the notion of past tense. Whether this omission is acceptable or not depends on the intended use.

The second sentence can be represented by

allow(x,H)

where x is a variable that can take any particular value. Conventionally, variables are denoted by lower case letters, and specific values, by using upper case as the first letter.

Sometimes, it is easy to determine that two assertions cannot both be true at same time [4], for example, L and ~L. In predicate logic, this matching process is more complicated since the arguments of the predicates must be considered. For example, man(John) and ~man(John) is a contradiction, while man(John) and ~man(Spot) is not. Thus, in order to determine contradictions, we need a matching procedure that compares two assertions and discovers whether there exists a set of substitutions that makes them identical. There is a straightforward recursive procedure, called the unification algorithm, that does just this.

The base idea of unification is very simple. To attempt to unify two literals (well-formed formulas converted to Clause Form, see [4]), we first check if their initial predicate symbols are the same. If so, we can proceed. Otherwise, there is no way they can by unified, regardless of their arguments. For example, the two literals

deny(John)

sleep(John)

cannot be unified. If the predicate symbols match, then we must check the arguments, one pair at a time. If the first matches, we can continue with the second, and so on. To test each argument pair, we can simply call the unification procedure recursively. The matching rules are simple. Different constants or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a predicate expression, with the restriction that the predicate expression must not contain any instances of the variable being matched.

The only complication in this procedure is that we must find a single, consistent substitution for the entire literal, not separate ones for each piece of it. To do this, we must take each substitution that we find and apply it to the remainder of the literals before we continue try to unify them. For example, suppose we want to unify the expressions

deny(x,x)

deny(y,z)

The two instances of deny match. Next we compare x and y, and decide that if we substitute y for x, they could match. We will write that substitution as

y/x

(Maybe it is easier to view this substitution as x:=y). Of course, we could have decided instead to use x/y, since they are both just dummy variable names. But now, if we simply continue and match x and z, we produce the substitution z/x. But we cannot substitute both y and z for x, so we have not produced a consistent substitution. What we need to do after finding the first substitution y/x is to make that substitution throughout the literals, giving

deny(y,y)

deny(y,z)

Now we can try to unify arguments y and z, which succeeds with the substitution z/y. The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found.

The object of the unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution, there are many. The procedure Unify(L1,L2) returns a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was found without any substitutions. The value FAIL signals that the unification procedure failed.

Algorithm: Unify(L1,L2)

1. if L1 or L2 is a variable or constant, then
   (a) if L1 and L2 are identical, then return NIL.
   (b) else if L1 is a variable, then if L1 occurs in L2 then return FAIL, else return {(L2/L1)}.
   (c) else if L2 is a variable then if L2 occurs in L1 then return FAIL, else return {(L1/L2)}.
   (d) else return FAIL.
2. if the initial predicate symbols in L1 and L2 are not identical, then return FAIL.
3. if L1 and L2 have different numbers of arguments, then return FAIL.
4. set SUBST to NIL. (At termination, SUBST contains all substitutions to unify L1 and L2).
5. for i:=1 to number of arguments in L1
   (a) call Unify with i-th argument of L1 and i-th argument of L2, placing the result in S.
   (b) if S=FAIL then return FAIL.
   (c) if S is not equal to NIL then
      i. apply S to the remainder of both L1 and L2.
      ii. SUBST:=APPEND(S,SUBST).
6. return SUBST.

Note: Steps 1(b) and 1(c) are to make sure that an expression involving a given variable is not unified with that variable. Suppose we were attempting to unify the expressions

f(x,x)

f(g(x),g(x))

If we accepted g(x) as a substitution for x, then we would have to substitute it for x in remainder of the expressions. But this leads to infinite recursion since it will never be possible to eliminate x.

We now have an easy way of determining that two predicate are contradictory - they are if one of them can be unified with the negation of the other. So for example, man(x) and ~man(Spot) are contradictory, since man(x) and man(Spot) can be unified. This corresponds to the intuition that man(x) cannot be true for all x if there is known to be some x, say Spot, for which man(x) is false.

Consistency checking of an access control list

The idea developed for predicate unification can serve as a method to check consistency among the statements in an access control list. We view each line in an access control list as a predicate such as deny(...) or allow(...). Because

the ACL is checked line by line, we can defined a set of statements P which initially consists of the first line. We check whether the second line is contradictory or redundant to P; if not, we add the line to P. That is, in each iteration, we check if one line is consistent with all of the previous lines, if so, we add this line to P. Specifically, for each line, the following steps are used:

Step 1. Unify the predicate (the line) with the predicates in P, considering all its arguments as constants (not to take other value). If the unification algorithm succeeds, the line is redundant. If the unification fails, the line is not redundant.

Step 2. Unify the negation of the predicate with the predicates in P, considering all its arguments as constants (not to take other value). If the unification algorithm succeeds, the line is contradictory; proceed to the next line. If unification fails, go to Step 3.

Step 3. Unify the negation of the predicate with the predicates in P. If the unification algorithm succeeds, then the line is possibly contradictory. If the unification fails, the line is definitely not contradictory. Proceed to the next line.

The following examples illustrate the consistency checking procedure:

Example 1: The access control list consists of:

deny(x,U)
deny(H,y)

In the iteration of checking deny(H,y) with P={deny(x,U)}, Step 1 fails because U cannot be unified with y (considered a constant). Thus the second line is not redundant. In Step 2, the unification fails with the negation of deny(H,y), so it is not contradictory.

Example 2: The access control list consists of:

deny(x,z)
deny(H,y)

In the iteration of checking deny(H,y) with P={deny(x,z)}, Step 1 succeeds, because a substitution is H/x, y/z. Therefore deny(H,y) is redundant. In Step 2, the unification fails with the negation of deny(H,y), so it is not contradictory.

Example 3: The access control list consists of:

deny(x,z)
allow(H,y)

In the iteration of checking allow(H,y) with P={deny(x,z)}, Step 1 fails, so there is no redundancy. Step 2 succeeds, because there is a substitution H/x, y/z to unify the negation of allow(H,y) which is deny(H,y) with P. So allow(H,y) is contradictory.

Example 4: The access control list consists of:

deny(x,U)
allow(H,y)

In the iteration of checking allow(H,y) with P={deny(x,U)}, Step 1 fails, so there is no redundancy. Step 2 fails also, by considering y a constant. Step 3 succeeds, because there is a substitution H/x, U/y to unify the negation of

allow(H,y) which is deny(H,y) with P. Therefore allow(H,y) is possibly contradictory to deny(x,U). Intuitively, the first predicate says that we deny user U no matter on which host he is. The second predicate says that we allow every one from host H. The dilemma is, what about user U from host H?

Example 5: The access control list consists of:

allow(H,U)

deny(x,U)

In the iteration of checking deny(x,U) with P={allow(H,U)}, Step 1 fails, so there is no redundancy. Step 2 fails also, by considering x a constant (H cannot be unified with x). Step 3 succeeds: there is a substitution H/x to unify the negation of deny(x,U) which is allow(x,U) with P. So deny(H,y) is possibly contradictory. Intuitively, if the purpose of this ACL is allow user U from host H and not any other hosts, the ACL is correct. Compare this with Example 4: if Step 3 succeeds, the ACL may be correct or contradictory.

5. Alarm Specifications

Alarms are actions triggered by certain events. Our proposed syntax for the specification of alarms is:

```
WHEN (alarm_condition) { action }
```

Actions may consist of commands and shell scripts. A network administrator can invoke a command file where he defines whatever he wants.

Some examples of this construct are:

```
WHEN ($DENY[$DESTINATION] > threshold & $ALERT_TIME == time_interval)
{ mail sysadm -s "intrusion" }
```

```
WHEN ($DENY[$SOURCE] > threshold & $ALERT_TIME == time_interval)
{ add2black source=$SOURCE }
```

Note that threshold specification is a typical way for real-time intrusion detection. A real-time system for intrusion detection is found in [3]. Recall that `$DENY[$DESTINATION]` and `$DENY[$SOURCE]` are the numbers of failures to connect the current destination and the source respectively. add2black can be a shell script like

```
echo $1 $2 | cat >> blacklist
kill -HUP ratchet
```

which adds the source to the blacklist and signals to "ratchet" to re-read the blacklist. We can see that several usual attack scenarios can be detected using this construct.

Attack scenarios

Suppose that several external hosts are allowed to access a single internal host, protected by a firewall (Figure 1). Depending on a attacker's ability, he can

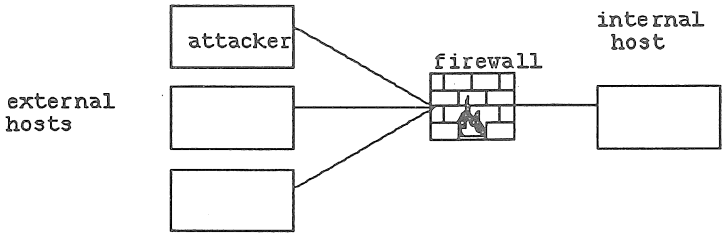

Figure 1 Attack on a single internal host

(1) attack from an external host without changing user name but with different passwords

(2) attack from an external host by changing his user name

(3) attack from different hosts at the same time

In a simplistic approach, one may define the following clauses:

```
WHEN ($DENY[$USER]>threshold & $ALERT_TIME==300)
{ add2black user=$USER }
WHEN ($DENY[$SOURCE]>threshold & $ALERT_TIME==300)
{ add2black source=$SOURCE }
WHEN ($DENY[$DESTINATION]>threshold & $ALERT_TIME==300)
{ add2black destination=$DESTINATION }
```

That is, whenever the number of failed attempts exceeds a given threshold, the user, the source, or the destination will be added to the blacklist. A better policy would be: In case (1), only the user is added to the blacklist, meanwhile other users on the same host should be allowed. In case (2), the external host where the attacker is on is added to the blacklist. In case (3), the destination should be protected by denying all the connections. (i.e., they would be added to the blacklist).

Using our construct, the following set of clauses will achieve the goal; we assume that the threshold is 5 failures during the last 300 seconds:

```
WHEN ($DENY[$USER]>5 & $ALERT_TIME==300)
{ add2black user=$USER }
WHEN ($DENY[$SOURCE]>5 & $DENY[$USER]<=5 & $ALERT_TIME==300)
{ add2black source=$SOURCE }
WHEN ($DENY[$DESTINATION]>5 & $DENY[$SOURCE]<=5 & $ALERT_TIME == 300)
{ add2black destination=$DESTINATION }
```

The first clause takes care of case (1) of the attack scenarios. In the second clause, `$DENY[$USER]<=5` means some other user has failed to connect from the same source, because the number of failures from this source exceeds 5. Therefore attack scenario (2) is assumed and the source will be added to the blacklist. Similarly, `$DENY[$SOURCE]<=5` means some other host has failed to connect to the destination, so attack scenario (3) is assumed. Now consider that we have several internal hosts as shown in Figure 2.

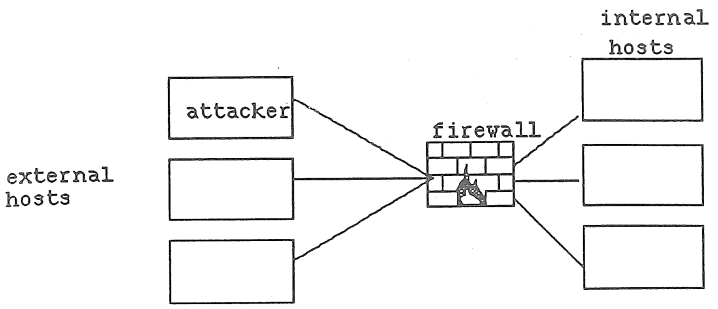

Figure 2 Attacks on several internal hosts

If the attacker tries to connect to several internal hosts without changing his user name, the first alarm clause will add him to the blacklist. If he tries to connect to several internal hosts changing his user name on a single host, the second alarm clause will add the source host to the blacklist. Finally, if he tries from different hosts changing his user name or if we have several attackers at same time, the denied connection numbers for the destinations will be high, which triggers the third alarm to add the destination to the blacklist.

6. Conclusion

We have defined a firewall access definition language to help in maintaining appropriate and easily understood and modifiable access restrictions. This provides ease of use, which in turn translates into making firewalls a more attractive approach to increasing network security. An added benefit of the approach is that consistency is checked automatically by the compiler; inconsistencies in the access definitions are often an indication of erroneous restriction formulation, which may result in reduced security.

7. References

[1] J. Bentley, "Little Languages", Communications of the ACM, vol. 29, no.8, pp. 711-721, August 1986.
[2] R. M. Kaplan, *Constructing Language Processors for Little Languages*, Wiley, 1994.
[3] T. F. Lunt and R. Jagannathan, "A Prototype Real-Time Intrusion-Detection Expert System", Proceedings of IEEE Symp. on Security and Privacy, pp. 59-66, 1988.
[4] E. Rich, E. and K. Knight, *Artificial Intelligence*, McGraw-Hill, 1991.
[5] Jianyu You and E. L. Leiss, "Firewall Monitoring", CLEI PANEL 96, Bogota, Colombia, June 3-7, 1996.
[6] http://www.socks.nec.com - Document for SOCKS and free software, directory: /